NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MODELING DATA RATE AGILITY IN THE IEEE 802.11a WIRELESS LOCAL AREA NETWORKING PROTOCOL

by

Bryan E. Braswell

March 2001

Thesis Advisor:	John McEachen
Second Reader:	Murali Tummala

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 2001

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE :
Modeling Data Rate Agility in the IEEE 802.11a Wireless Local Area Networking Protocol

5. FUNDING NUMBERS

6. AUTHOR(S)
Braswell, Bryan E.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING /MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*
The IEEE 802.11a high-speed wireless local area networking (WLAN) protocol does not specify a mechanism for dynamically altering network data rates based on changing link conditions. This thesis first presents a baseline software model of the 802.11a protocol developed using the OPNET simulation tool. The model includes both the medium access control (MAC) and physical (PHY) layers of the standard. Two data rate agility mechanisms are then proposed and analyzed using the model. An infrastructure WLAN implementation of the baseline model is first simulated under standard network conditions to verify its operational characteristics and the results are presented. The model is then used to simulate two data rate agility mechanisms, one based on the link signal-to-noise ratio (SNR) and the other based on the frame loss rate at the transmitting station. Each technique is simulated using an infrastructure WLAN consisting of a fixed access point and a mobile workstation operating with standard network traffic loads. The results indicate that the link SNR is a better decision criterion for data rate agility than the frame loss rate. The design and methodology of this analysis provides insight into dynamic rate agility mechanisms and the criteria that may be used in developing future 802.11a-compliant hardware implementations.

14. SUBJECT TERMS
Wireless Networking, Protocol Analysis, 802.11a, Orthogonal Frequency Division Multiplexing, Medium Access Control, Rate Agility, Optimum Network Performance

15. NUMBER OF PAGES
224

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

MODELING DATA RATE AGILITY IN THE IEEE 802.11a WIRELESS LOCAL AREA NETWORKING PROTOCOL

Bryan E. Braswell
Lieutenant, United States Navy
B.A., University of Virginia, 1993

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2001

Author: Bryan E. Braswell
Bryan E. Braswell

Approved by:
John McEachen, Thesis Advisor

Murali Tummala, Second Reader

Jeffrey B. Knorr, Chairman
Department of Electrical Engineering and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The IEEE 802.11a high-speed wireless local area networking (WLAN) protocol does not specify a mechanism for dynamically altering network data rates based on changing link conditions. This thesis first presents a baseline software model of the 802.11a protocol developed using the OPNET simulation tool. The model includes both the medium access control (MAC) and physical (PHY) layers of the standard. Two data rate agility mechanisms are then proposed and analyzed using the model. An infrastructure WLAN implementation of the baseline model is first simulated under standard network conditions to verify its operational characteristics and the results are presented. The model is then used to simulate two data rate agility mechanisms, one based on the link signal-to-noise ratio (SNR) and the other based on the frame loss rate at the transmitting station. Each technique is simulated using an infrastructure WLAN consisting of a fixed access point and a mobile workstation operating with standard network traffic loads. The results indicate that the link SNR is a better decision criterion for data rate agility than the frame loss rate. The design and methodology of this analysis provides insight into dynamic rate agility mechanisms and the criteria that may be used in developing future 802.11a-compliant hardware implementations.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. OBJECTIVE 2
C. RELATED WORK 3
D. THESIS ORGANIZATION 4
II. THE IEEE 802.11A PROTOCOL 5
A. NEXT GENERATION WLAN PROTOCOLS 6
B. THE 802.11A MAC LAYER 9
C. THE 802.11A PHYSICAL LAYER 15
III. IEEE 802.11A PROTOCOL MODEL 21
A. OPNET AND THE 802.11 STANDARD MODEL 21
B. THE 802.11A BASELINE MODEL 23
1. The 802.11a Model MAC Layer 26
2. The 802.11a Model PHY Layer 30
C. 802.11A BASELINE MODEL SIMULATION RESULTS 35
IV. DATA RATE AGILITY AND THE 802.11A BASELINE MODEL 45
A. DATA RATE AGILITY MECHANISMS 45
1. Rate Agility Based on Link SNR 46
2. Rate Agility Based on the Frame Loss Rate 49
B. RATE AGILITY MECHANISM SIMULATION RESULTS 55
V. CONCLUSIONS AND RECOMMENDATIONS 63
A. CONCLUSIONS 63
B. RECOMMENDATIONS 64
APPENDIX A. *WLAN_MAC_11A* PROCESS MODEL OPNET CODE 67
APPENDIX B. SNR-BASED RATE AGILITY OPNET CODE 149
APPENDIX C. PACKET LOSS RATE-BASED RATE AGILITY OPNET CODE 153
APPENDIX D. *DRA_SNR_11A* PIPELINE STAGE 161
APPENDIX E. *DRA_TXDEL_11A* PIPELINE STAGE 163
LIST OF REFERENCES 165
INITIAL DISTRIBUTION LIST 167

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

Figure 1. 802.11a and the OSI and TCP/IP Models 5
Figure 2. Infrastructure and Independent WLAN BSSs 10
Figure 3. Timing Relationships in the 802.11 Standards 13
Figure 4. Timing in a Data Transmission Scenario 14
Figure 5. The 802.11a MPDU 15
Figure 6. The 802.11a PPDU 18
Figure 7. OPNET Design Environment 22
Figure 8. An 802.11a WLAN in OPNET 24
Figure 9. The 802.11a Model STA Node Model 25
Figure 10. The 802.11a Model AP Node Model 26
Figure 11. The *wlan_mac_11a* Process Model 27
Figure 12. The *wlan_data_802_11a* Packet Format 30
Figure 13. The *wlan_control_802_11a* Packet Format 30
Figure 14. BER versus E_b/N_o Curves for OFDM in AWGN 33
Figure 15. Simulated 802.11a Network Environment 36
Figure 16. The Simulated 802.11a WLAN BSS 37
Figure 17. 802.11a Model Simulation Network Load Configuration 38
Figure 18. Total Load on the Simulated WLAN 41
Figure 19. Individual Load Values for the AP and STAs 41
Figure 20. Simulated Medium Access Delay and Packet Delay 42
Figure 21. Simulated Link SNRs 43
Figure 22. Simulated Packet Loss Rate in High Traffic Conditions 52
Figure 23. Simulated Packet Loss Rate in Low Traffic Conditions 53
Figure 24. Rate Agility Simulation Environment 56
Figure 25. Simulated Data Rates with SNR-Based Rate Agility 57
Figure 26. Simulated Link BERs with SNR-Based Rate Agility 57
Figure 27. Simulated Link SNRs with SNR-Based Rate Agility 58
Figure 28. Simulated Data Rates with Frame Loss Rate-Based Rate Agility 60
Figure 29. Data Rates for Both Mechanisms 61
Figure 30. The *wlan_mac_11a* Process Model 67

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. International 5 GHz WLAN Standards 8
Table 2. Multimedia Applications and Associated Data Rates 8
Table 3. 802.11a IFS and Slot Time Values and Definitions 13
Table 4. Channelization in the 802.11a Standard 16
Table 5. 802.11a Maximum Output Power Levels 17
Table 6. Coding and Modulation in the 802.11a Standard 17
Table 7. User-Defined *Wireless LAN* Parameters 28
Table 8. Attributes of the *wlan_port_tx* and *wlan_port_rx* Node Objects 31
Table 9. Simulated WLAN Traffic Profiles 39
Table 10. WLAN Attributes and Simulation Characteristics 40
Table 11. SNR Thresholds for Rate Agility 49
Table 12. 802.11a Ranges with SNR-Based Rate Agility 59
Table 13. Nominal Ranges of the Lucent ORiNOCO PC Card 59

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS

ACK	Acknowledgement Frame
AP	Access Point
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
BRAN	Broadband Radio Access Networks
BSS	Basic Service Set
COFDM	Coded Orthogonal Frequency Division Multiplexing
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
CTS	Clear To Send Frame
DAB	Digital Audio Broadcasting
DCF	Distributed Coordination Function
DIFS	Distributed Coordination Function Interframe Space
DVB	Digital Video Broadcasting
EIFS	Extended Interframe Space
ETSI	European Telecommunications Standards Institute
FCC	Federal Communications Commission
FCS	Frame Check Sequence
FDM	Frequency Division Multiplexing
FEC	Forward Error Correction
FTP	File Transfer Protocol
HDTV	High Definition Television
HIPERLAN	High Performance Radio Local Area Network
HTTP	Hypertext Transfer Protocol
ICI	Intercarrier Interference
IEEE	Institute of Electrical and Electronics Engineers
IFS	Interframe Space
ISI	Intersymbol Interference
ISM	Industrial, Scientific, and Medical
MAC	Medium Access Control
MMAC	Multimedia Mobile Access Communications
MPDU	Medium Access Control Protocol Data Unit

NAV	Network Allocation Vector
OEM	Other Equipment Manufacturer
OFDM	Orthogonal Frequency Division Multiplexing
OPNET	Optimum Network Performance
OSI	Open System Interconnectivity
PCF	Point Coordination Functions
PHY	Physical Layer
PIFS	Point Coordination Function Interframe Space
PLCP	Physical Layer Convergence Protocol
PPDU	Physical Layer Convergence Protocol Protocol Data Unit
PSDU	Physical Layer Convergence Protocol Service Data Unit
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature Phase Shift Keying
RF	Radio Frequency
RTS	Request To Send Frame
SIFS	Short Interframe Space
SNR	Signal-To-Noise Ratio
STA	Station
STD	State Transition Diagram
TCP/IP	Transport Control Protocol/Internet Protocol
UDP	User Datagram Protocol
UNII	Unlicensed National Information Infrastructure
VoIP	Voice Over Internet Protocol
WAN	Wide Area Network
WATM	Wireless Asynchronous Transfer Mode
WLAN	Wireless Local Area Network

EXECUTIVE SUMMARY

The Institute for Electrical and Electronics Engineers (IEEE) 802.11a wireless local area networking (WLAN) standard presents office, campus, and home networking consumers with the first viable wireless alternative to wired networks that can support the simultaneous use of high data rate applications in a mobile, multi-user environment. The 802.11a protocol standardizes both the medium access control (MAC) and the physical (PHY) layers. 802.11a-compliant WLANs will be able to support raw data rates ranging from 6 to 54 Mbps using a distributed Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) MAC scheme in conjunction with a PHY layer based on the use of Orthogonal Frequency Division Multiplexing (OFDM).

The 802.11a specification promises to deliver WLAN ranges and power levels commensurate with those of WLANs based on the 802.11 and 802.11b standards. Like the original 802.11 specification, the 802.11a addendum does not specify a mechanism through which a WLAN implementation should dynamically alter its data rates if the preset data rate is not achievable in a given link environment. The 802.11 standards explicitly address data rate agility insofar as they state that dynamic rate switching is allowed for, although specific techniques are beyond the scope of the protocol.

This thesis first presents a model of the 802.11a protocol developed using the Optimum Network Performance (OPNET) network modeling and simulation tool. The model emulates both the MAC and PHY layers of the standard. Simulation results obtained using the model are presented as a measure of its validity. Two dynamic data rate agility mechanisms are then proposed and analyzed using the OPNET 802.11a model. The first implements rate agility in a WLAN based on the instantaneous link SNR as measured at the PHY layer of the receiving station while the second technique uses the frame loss rate at the MAC layer of the transmitting station to achieve dynamic data rate agility. The goal is to both compare dynamic data rate mechanisms that target separate layers of the protocol stack and present a methodology for analyzing rate agility in 802.11a-compliant WLANs using OPNET.

The results obtained during simulations conducted using both mechanisms indicate that the link SNR is a better criterion than the packet loss rate upon which to base dynamic data rate agility decisions in IEEE 802.11a-compliant WLANs. The SNR-based mechanism achieved a higher mean data rate over the course of the simulation and exhibited smoother data rate transitions with less oscillation between rates. The mechanism based on frame loss rates was characterized by highly variable data rates and a lower mean data rate. The general trends obtained using the frame loss rate-based mechanism indicate that the frame loss rate is a good measure of the link quality; however, the link SNR proved to be a far better indicator.

The overall design and methodology of this analysis provides insight into dynamic rate agility mechanisms and the criteria for rate agility that may be used in developing future 802.11a-compliant hardware implementations.

ACKNOWLEDGMENT

This thesis is dedicated both to my wife Tracy and to my parents. A lifetime of support and encouragement from my parents led me to graduate school while Tracy's constant love and understanding ensured my success here. I am forever indebted to them.

My sincerest gratitude goes to my advisor and mentor, Dr. John McEachen, for his guidance, direction, and support. His supervision and seemingly limitless knowledge of networking all but guaranteed the success of the model and accompanying analysis presented in this thesis. My appreciation also goes to Dr. Murali Tummala, whose superior instructional abilities first led me to the study of networking and have kept me there ever since.

Finally, I would like to thank Dr. Sunghyun Choi of Philips Research Labs. A number of the MAC layer features of the wireless network model found herein were adapted from his group's 802.11a OPNET model work. His collaboration and thoughts concerning the simulation of 802.11a were invaluable.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

This thesis presents a wireless local area network model based on the Institute of Electrical and Electronics Engineers (IEEE) 802.11a protocol developed using the Optimum Network Performance (OPNET) simulation tool. The model incorporates features of the 802.11a standard that were developed in OPNET by Dr. Sunghyun Choi of Philips Research Labs, and it comprehensively models both the medium access control (MAC) and the physical (PHY) layers of the 802.11a protocol [1]. Some simulation results obtained using the model are presented as a measure of its validity.

A dynamic data rate agility mechanism was added to the baseline model to explore the criteria by which an 802.11a-based wireless local area network (WLAN) might dynamically alter its link data rates during operation. Like the original 802.11 specification, the 802.11a addendum does not specify a mechanism through which a WLAN implementation should dynamically alter its data rates if the preset data rate is not achievable in a given set of link conditions. The 802.11 standards explicitly address data rate agility insofar as they state that dynamic rate switching is allowed, although specific techniques are not delineated [2, 3, 4]. The rate agility mechanisms presented herein offer a methodology for examining and comparing several different criteria upon which a decision to switch data rates may be based. Specifically, rate-switching mechanisms based on link signal-to-noise ratios (SNRs) and on frame loss rates are examined and the results presented.

A. BACKGROUND

Wireless networking technologies have evolved from disparate proprietary implementations first conceived in the late 1980s and early 1990s to a set of overlapping global standards. Those early wireless networking realizations were designed for a limited number of specific applications, such as inventory control and shipment tracking in a warehouse-like environment. Throughout the 1990s, as international government and commercial reliance on wired internetworking grew, so too did the emphasis on mobility and the development of wireless networking standards. Today there are a number of approved international standards that will allow high-speed wireless networks to compete effectively with their wired counterparts. The most comprehensive and well developed of these standards is the IEEE 802.11a protocol.

In June of 1997 the IEEE approved the 802.11 WLAN standard [2]. The initial 802.11 protocol was designed to provide a standard for high data rate (i.e., up to 2 Mbps) WLAN connectivity in any campus, office, or home environment as well as in other more specialized settings. The European Telecommunications Standards Institute (ETSI) concurrently developed the High-Performance Radio LAN (HIPERLAN) protocol, also designed to provide standardized high data rate WLAN systems [5]. Soon thereafter a number of WLAN implementations based on the IEEE 802.11 protocol were developed and fielded by companies like Lucent, Aironet and Bay Networks, among others. These WLANs eventually obtained a small but solid share of the campus, office and home networking market.

The IEEE approved the 802.11b addendum to the original 802.11 specification that allowed for wireless networking at data rates of up to 11 Mbps a year later [3]. This addendum to the standard was rapidly included in commercial systems, allowing for even greater commercial adoption of WLAN implementations. Consumer demand for both mobility and high data rate multimedia applications such as video teleconferencing, streaming video, and voice over IP (VoIP) was growing. IEEE 802.11b-based networks providing (at most) 11 Mbps connectivity, although more capable than their predecessors, are not able to handle the strenuous traffic load imposed given the simultaneous use of multimedia applications in a multi-user WLAN environment.

To support the consumer demand for mobility and low latency, high data rate communications, the next generation of WLAN standards has emerged. Foremost among these is the 802.11a addendum to the IEEE 802.11 standard, with others being the European HIPERLAN/2 standard and the Japanese Multimedia Mobile Access Communications (MMAC) protocol in the 5 GHz band [6]. The IEEE 802.11a standard has received the most attention; due largely in part to the fact that the only fielded WLAN implementations available today are based largely on the 802.11 protocol family. The 802.11a protocol specifies operation in the 5 GHz band, utilizes orthogonal frequency division multiplexing (OFDM) in the PHY layer and provides for data rates ranging from 6 to 54 Mbps [4]. These data rates are clearly capable of supporting high traffic applications in a mobile, multi-user WLAN environment.

B. OBJECTIVE

The performance of the 802.11a protocol has not been extensively analyzed as it was only recently approved and there are no exiting commercial WLAN implementations

utilizing OFDM in the PHY layer. The first goal of this thesis is to develop a model of the 802.11a protocol using the OPNET modeling tool that incorporates both the MAC and the PHY layers of the standard. The model can then be used for further research that specifically targets either the MAC layer or PHY layer of the protocol or concerns the performance of 802.11a as a whole. There are a number of possible approaches to modeling 802.11a in OPNET, especially at the PHY layer. The model outlined herein presents one technique for modeling 802.11a.

The second objective of this thesis is to utilize the OPNET 802.11a model to analyze two dynamic data rate agility mechanisms. The two mechanisms are first presented and then their performance is compared using the model. The goals here are twofold: to present the performance analysis results obtained using each rate agility mechanism and to present a new research methodology for analyzing hypothesized data rate agility mechanisms. The use of the OPNET simulation tool in conjunction with the 802.11a model to study dynamic rate agility mechanisms will provide insight into the rate agility criteria that may be used when developing 802.11a-compliant WLANs

C. RELATED WORK

There are a number of ongoing efforts to develop models of the 802.11a protocol using OPNET. One such research project is underway at Philips Research Labs in New York. A number of MAC layer features of the model they are developing are included in the model presented in this thesis [1]. Moreover, throughout the course of the design and construction of the model outlined here, the author corresponded with a number of researchers also involved with the development of OPNET 802.11a protocol models. Active OPNET 802.11a modeling efforts are underway in universities and companies in Mexico, Japan, and the Netherlands to name a few. With the exception of the Philips Research Labs 802.11a MAC layer model, none of the models the author has been exposed to have yet been completed or used in active simulations.

The use of OPNET to simulate and analyze dynamic data rate agility mechanisms in 802.11a-compliant WLANs is a new research methodology. The research literature on rate agility mechanisms in standardized WLANs is extremely sparse, while implementation-specific details concerning rate agility techniques utilized in fielded 802.11- and 802.11b-compliant WLANs are proprietary and are unavailable to the author. Hardware vendors currently developing 802.11a-compliant products are still analyzing rate agility mechanisms and addressing the trade-offs associated with implementing rate

agility at the MAC and PHY layers, but again, their work is proprietary and is unavailable. Since the 802.11a standard does not specify a rate agility mechanism, any agility techniques developed by WLAN vendors will be proprietary in nature. Accordingly, the research methodology presented herein is a new approach to the 802.11a-compliant WLAN engineering issues associated with determining the optimum criteria for dynamic rate agility.

D. THESIS ORGANIZATION

This chapter has provided background information concerning the 802.11a protocol and its role in wireless networking. The objectives of this thesis were also presented along with a survey of current efforts in modeling 802.11a and the use of an 802.11a model to analyze dynamic data rate agility. In the next chapter, the important elements of the 802.11a protocol are outlined, to include both the MAC and PHY layers. The specifics of the baseline 802.11a model are then presented in Chapter III within the framework of the OPNET modeling and simulation tool. Simulation results obtained using the baseline model are provided as a measure of the model's validity. Chapter IV presents the data rate agility mechanisms added to the baseline 802.11a model along with a comparison of their rate switching criteria. Conclusions and recommendations are then included in the final chapter. Appendix A lists the code of the *wlan_mac_11a* OPNET process model for the 802.11a baseline model MAC, Appendices B and C outline the code changes required to implement the two data rate agility mechanisms, and Appendices D and E provide the two new OPNET pipeline stages required to support the 802.11a models.

II. THE IEEE 802.11a PROTOCOL

The 802.11a addendum to the original 802.11 standard presents office, campus, and home networking consumers with the first viable wireless alternative to wired networks that can support the simultaneous use of high data rate applications in a multi-user environment. 802.11a shares a number of features with the original 802.11 standard; however, its PHY layer is completely different from that of both 802.11 and 802.11b and is able to deliver data rates of up to 54 Mbps. The 802.11a standard therefore allows for robust, high data rate wireless connectivity in a variety of network environments.

The standardization scope of the 802.11 protocol family (i.e., the original 802.11 standard and the 802.11a and 802.11b addendums) includes both a portion of the Data Link Layer and the Physical Layer of the Open System Interconnectivity (OSI) layered model and the Network Access Layer of the TCP/IP protocol suite's layered model (see Figure 1). The 802.11 protocol family therefore standardizes the MAC and PHY layers of the WLAN. With very minor differences the MAC layer of each 802.11 specification is essentially identical. Furthermore, the PHY layer of the original 802.11 standard and the 802.11b addendum are, with a few exceptions, very similar in that each uses spread spectrum transmission techniques.

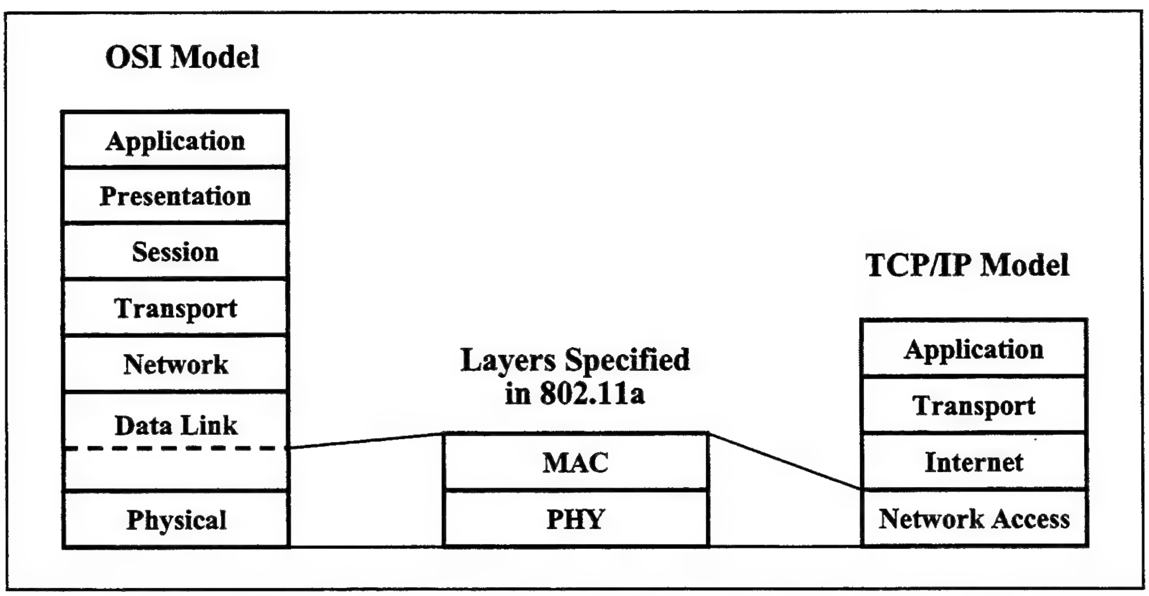

Figure 1. 802.11a and the OSI and TCP/IP Models (After Refs. [7, 8]).

The 802.11a standard, however, uses a completely different PHY layer encoding scheme that operates in a higher frequency band. 802.11a was designed from the start to operate in the 5 GHZ band vice the 2.4 GHz band like 802.11 and 802.11b. This followed the Federal Communications Commission's (FCC's) 1997 decision to allocate 300 MHz of radio frequency (RF) spectrum for unlicensed operation in the new 5 GHz Unlicensed National Information Infrastructure (UNII) band [6]. WLAN implementations currently operating in the 2.4 GHz band have to compete for the same RF spectrum with cordless phones, microwaves, and other WLAN devices while the newly available 5 GHz band offers a relatively interference-free spectrum [9]. To take advantage of this higher frequency band other alterations to the PHY layer were required to offset the decreases in range and higher power requirements that would have accompanied the frequency band change alone. At the PHY layer 802.11a uses an adaptation of OFDM for encoding and transmission called coded OFDM (COFDM). COFDM is a frequency division multiplexing (FDM) multi-carrier communications scheme that includes the appplication of convolutional coding to achieve higher data throughput rates. COFDM will be covered in greater detail in the PHY layer subsection of this chapter. Before taking a closer look at the specification itself, 802.11a's role within the framework of an emerging group of new global WLAN standards will first be addressed.

A. NEXT GENERATION WLAN PROTOCOLS

The IEEE 802.11a protocol is only one of a number of global WLAN standards that have been developed to support mobile, high data rate wireless networking. As discussed in Chapter I, the first wireless networks were proprietary implementations designed to operate as stand-alone systems. The 802.11 standard was the first to codify a set of guidelines within which a WLAN should be designed if the vendor sought interoperability with other WLAN systems. At the same time, the ETSI was developing the HIPERLAN standard, designed for operation in the 5 GHz band [5]. In the years following the release of the 802.11 standard, every fielded commercial implementation (both in the U.S. and in Eurpoe) was based on the 802.11 specification primarily because 802.11 compliant systems achieved a foothold in the global marketplace before the final HIPERLAN standard was ever released. In addition, the 2.4 GHz industrial, scientific, and medical (ISM) RF band within which 802.11 (and 802.11b) WLANs operate is

readily available internationally, so consumers could purchase and field 802.11 WLANs without any serious regulatory concerns.

Consumer demand for a combination of mobility and multimedia applications in conjunction with the FCC's ruling drove the emergence of the IEEE 802.11a working group. The group rapidly adopted OFDM as the standard's underlying PHY layer technology as OFDM could clearly provide the requisite data rates. The ETSI's Broadband Radio Access Networks (BRAN) HIPERLAN working group was simultaneously developing HIPERLAN/2, the 5 GHz follow on to the HIPERLAN standard. Soon after OFDM was chosen by the IEEE 802.11a working group ETSI BRAN also chose OFDM as the PHY layer technology for HIPERLAN/2 [10]. Shortly thereafter, the Japanese also adopted OFDM for their 5 GHz MMAC standard. Essentially these standards bodies collaborated to create, to some degree, a global WLAN PHY layer standard.

Despite their similarities, there are still a number of differences between the standards below the surface. The salient features of the three international 5 GHz WLAN standards are provided in Table 1. Their data rates vary, ranging from 36 Mbps (MMAC) to 54 Mbps (802.11a and HIPERLAN/2). This range of data rates is ample enough to support even the most demanding multimedia applications, such as High Definition Television (HDTV), which requires support for at least 20 Mbps. Other multimedia applications that are supported by 802.11a, HIPERLAN/2 and MMAC and their associated traffic loads are shown in Table 2. At the PHY layer, the specific RF bands and power requirements in the 5 GHz range differ due to varying international regulatory restrictions and this, in turn, affects channelization and data rates. HIPERLAN/2 also utilizes a connection-oriented MAC that is essentially a wireless asynchronous transfer mode (WATM) call set-up scheme that promises interoperability with IP-based networks [11]. In short, HIPERLAN/2 has a redesigned and very complex MAC layer, one that has never been commercially implemented.

Attribute	Standard		
	802.11a	**HIPERLAN/2**	**MMAC**
Location	United States	Europe	Japan
Governing Body	IEEE	ETSI	Ministry of Post and Telecommunications
Frequency Bands	5.15 – 5.25 GHz 5.25 – 5.35 GHz 5.725 – 5.825 GHz	5.15 – 5.35 GHz 5.470 – 5.725 GHz	5.15 – 5.35 GHz
Supported Data Rates	6, 9, 12, 18, 24, 36, 48, 54 Mbps	6, 9, 12, 18, 27, 36, 54 Mbps	6, 12, 27, 36 Mbps
PHY Layer	OFDM	OFDM	OFDM
MAC Layer	CSMA/CA or PCF	TDMA/TDD with QoS Support	CSMA/CA or TDMA/TDD

Table 1. International 5 GHz WLAN Standards.

Application	Technique	Required Data Rate
Streaming Video	MPEG-4	0.005 – 10 Mbps
Broadcast Quality Video	MPEG-2	2 – 4 Mbps
HDTV	MPEG-2	25 – 34 Mbps
Streaming Audio	MPEG Layer 3 (MP3)	0.032 – 0.32 Mbps
Studio Quality Sound	MPEG with FFT	0.384 Mbps
Standard Voice	G.711 PCM	0.064 Mbps
DSL	ADSL	1.5 – 9 Mbps

Table 2. Multimedia Applications and Associated Data Rates (After Ref. [12]).

The 802.11a specification is the only one of the three for which a basic hardware implementation has been developed commercially. In September of 2000 Radiata Communications, Inc. announced that it had developed the first commercial implementation of the 802.11a protocol in the form of a chipset that includes both a modem chip and a transceiver chip [13]. Atheros Communications, Inc. has also released a similar chipset to implement 802.11a along with a proprietary protocol allowing for a 72 Mbps data rate [14]. Both chipsets are constructed using standard-process CMOS and

each is expected to retail for approximately $35.00. No OEM vendors have yet fielded a WLAN that implements either of these 802.11a-compliant chipsets.

The 802.11a protocol is clearly well positioned to succeed in the near-term as the predominant high data rate WLAN standard, not only because of its status on the market today but also because it shares the entirety of its MAC layer with already fielded 802.11-compliant products. As a result, there is a greater degree of familiarity with the 802.11 protocol family MAC and transitioning to new 802.11a-based products will require less cost and instructional overhead. The 802.11a MAC will be outlined in the next subsection and the PHY layer will be described in the subsequent subsection.

B. THE 802.11A MAC LAYER

The WLAN MAC layer is essentially identical across each member of the 802.11 protocol family. The 802.11a MAC will be addressed here to the extent that it applies to the model presented in this thesis. Accordingly, the major tenants of the 802.11a MAC will be covered; however, some minor details will be omitted for the sake of brevity. The 802.11 standard itself and references [15] and [16] are excellent sources of information on the 802.11 family MAC layer. The primary difference between the members of the 802.11 family of MACs is obviously the set of supported data rates and mandatory rates, but the rules governing the usage of those rates remain essentially the same. The 802.11a protocol allows for data rates of 6, 9, 12, 18, 24, 36, 48, and 54 Mbps. Of those, the 6, 12, and 24 Mbps speeds comprise the mandatory rate set, meaning that every 802.11a-compliant WLAN implementation must, at a minimum, support both transmission and reception at those data rates. Note that both the Radiata, Inc. and Atheros, Inc. chipsets support each of the delineated 802.11a data rates [13, 14].

In general, the groups of terminals that comprise a single 802.11 WLAN segment are referred to as a basic service set (BSS). The 802.11 MAC is designed to operate in one of two general network architectures: the infrastructure BSS or the independent BSS. An independent BSS is a WLAN that consists of mobile peer stations (STAs) that operate in an ad-hoc manner without any external connectivity. The infrastructure BSS is one in which the mobile STAs all communicate through a single fixed access point (AP) that is wired to an external network. Figure 2 illustrates the differences between the two. The vast majority of fielded 802.11 and 802.11b implementations are infrastructure BSSs since the goal in the home, office, or campus environment is often to use the WLAN to allow for mobility while bridging to a wired external network. The model presented in

Chapter III is that of an 802.11a infrastructure WLAN. Within a BSS, the 802.11 protocol standardizes both the manner in which a wireless STA joins, or associates with, the BSS and the authentication and encryption procedures used to maintain security within the WLAN. Neither feature is modeled here; therefore the details of those processes will not be discussed.

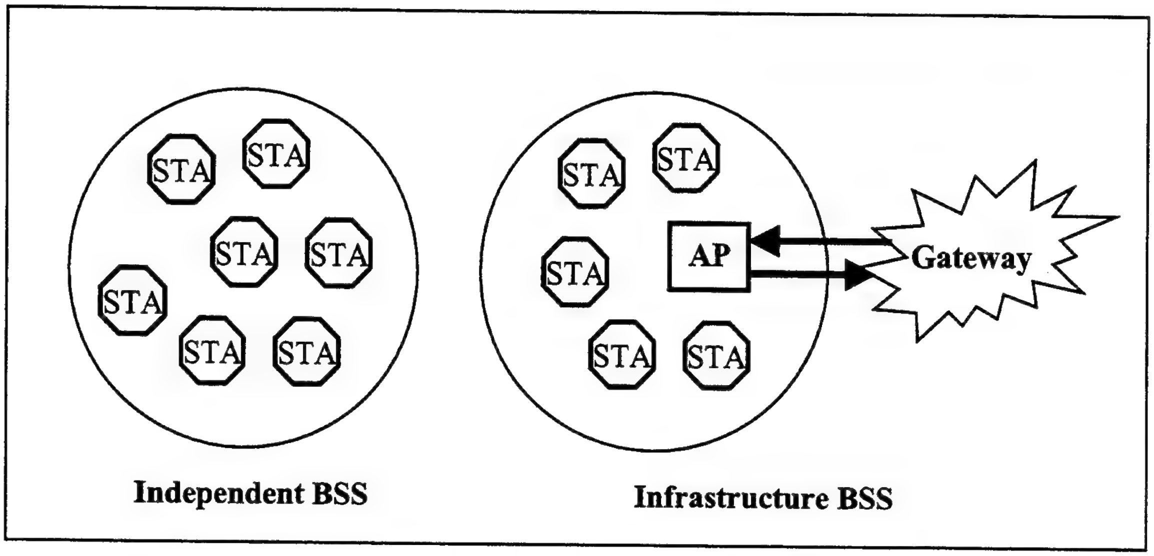

Figure 2. Infrastructure and Independent WLAN BSSs.

The primary function of the MAC layer is, as its name suggests, the control of access to the RF medium by each node in a BSS. The 802.11 protocol family allows for two access schemes: the distributed coordination function (DCF) and the point coordination function (PCF). The PCF is a medium reservation scheme applied only to infrastructure BSSs, as it consists of a polling cycle whereby the AP polls each mobile member of the BSS to both send and receive traffic in a time slot reserved by the AP. The PCF is best employed in a WLAN with few users and when each user is dealing with data that requires a very low latency. Accordingly, the PCF is rarely used in practice, and (per the standard) is an optional medium access technique in 802.11-compliant WLANs.

All 802.11-compliant WLANs must be able to employ the DCF access scheme where control of access to the RF medium is distributed amongst each STA in the BSS. STAs implement the DCF using the Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) algorithm. This access mechanism is similar to the one employed in conventional 802.3 Ethernet LANs (e.g., CSMA with collision detection), however collision detection is impossible in a WLAN environment since wireless nodes cannot simultaneously transmit and sense the medium. As a result, 802.11 STAs use

collision avoidance techniques to minimize collisions on the medium and the resulting cost in terms of network overhead. This collision avoidance mechanism is realized through the use of physical carrier sensing at each STA. Each STA physically senses the RF medium to determine if it is busy (i.e., if another STA is transmitting) or idle. If idle, the STA may transmit, but if the medium is busy the station waits until the medium becomes idle and then "backs off," or waits, a random amount of time before beginning transmission. Once the frame has been sent and if no collisions occurred during transmission, the receiving STA sends an acknowledgement (ACK) frame to the transmitting STA to confirm that the data was received successfully.

Physical carrier sensing, random back off and the use of ACK frames combine to reduce the delay and overhead associated with multi-user communications on a shared medium. For the process to function optimally, each STA in the BSS must be within communications range of each other and not just the AP. If mobile STAs are out of range of each other but each within range of the AP, then the physical carrier sensing mechanism will not be effective in avoiding collisions at the AP's receiver and the WLAN's performance will suffer as a result. This is commonly referred to as the "Hidden Node Problem," and the 802.11 MAC has included an additional, optional technique to address it using a virtual carrier sensing mechanism.

Virtual carrier sensing enables a STA to reserve the RF medium (BSS-wide) for a specific amount of time so as to prevent other STAs that may be "hidden" from transmitting simultaneously. When this mechanism is implemented a STA wishing to transmit sends a Request To Send (RTS) frame to the AP asking for permission to transmit (i.e., reserve the medium) for a given amount of time, as determined by the amount of data the STA has to send. If the medium is free the AP responds with a Clear To Send (CTS) frame, which serves to inform the requesting STA that it may transmit. When responding with a CTS, the AP includes the duration of the impending transmission within the body of the CTS frame so that all STAs in the BSS will be exposed to the length of time that the medium will be busy. When a non-transmitting STA receives the CTS frame from the AP it sets a timer, called the Network Allocation Vector (NAV), that tracks the length of time that the medium is expected to be busy. A STA's NAV therefore, based on the observation of a RTS/CTS exchange, provides the virtual carrier sensing capability.

The RTS/CTS functionality has the potential to either increase or decrease the overall performance of a WLAN. The trade-off is between the overhead associated with the RTS/CTS exchange and the performance degradation due to hidden node collisions

when RTS/CTS is disabled. The sizes of the data frames that are transmitted by the STAs tend to be the deciding factor in terms of efficiency. RTS/CTS benefits performance when the data frames are larger and the likelihood of a collision on the medium is increased. When the frames are small, the decreased chance of a collision outweighs the benefits of employing RTS/CTS. Accordingly, a user-defined frame length threshold is specified for each WLAN above which the RTS/CTS mechanism is enabled. WLAN performance can also be enhanced through the use of an optional frame fragmentation mechanism. Longer frames tend to have higher error rates therefore the transmission of long frames increases both the number of required retransmissions and the amount of data that is dropped. To combat this inefficiency the 802.11 standards specify a mechanism for fragmenting frames when their length is above a user-defined threshold. When a frame is fragmented each segment is transmitted as if it were a separate frame, while the fragments are identified at the destination using various fields in the packet header.

For both the CSMA/CA and RTS/CTS mechanisms to function properly, timing is obviously very important. Proper timing is accomplished through the use of four different interframe spaces (IFSs) and the slot time, defined below in Table 3 and illustrated in Figure 3. Note the slot time's importance in determining the back off period used by each station during the contention window, or the window during which each station vies for use of the RF medium. The IFSs and the slot time are selected based on the PHY layer characteristics, so the values shown in Table 3 are particular to the 802.11a protocol. A typical 802.11 WLAN MAC-level transmission scenario is depicted in Figure 4, where the timing relationships among the transmitting, receiving, and other BSS STAs are clearly shown.

Timing Parameter	Value	Description
SIFS	16 μs	Short IFS. The time required for a transceiver to alternate between transmit and receive modes. Used with ACK and CTS frames.
Minimum CW	15	Minimum Contention Window Size.
Maximum CW	1023	Maximum Contention Window Size.
Slot Time	9 μs	Used to determine the random back off time, given by: *Backoff = Random x Slot Time* where the random number is from the contention window interval.
DIFS	34 μs	DCF IFS. Used in transmitting data and management frames.
EIFS	94 μs	Extended IFS. Used when a frame is received with an incorrect FCS field.

Table 3. 802.11a IFS and Slot TimeValues and Definitions.

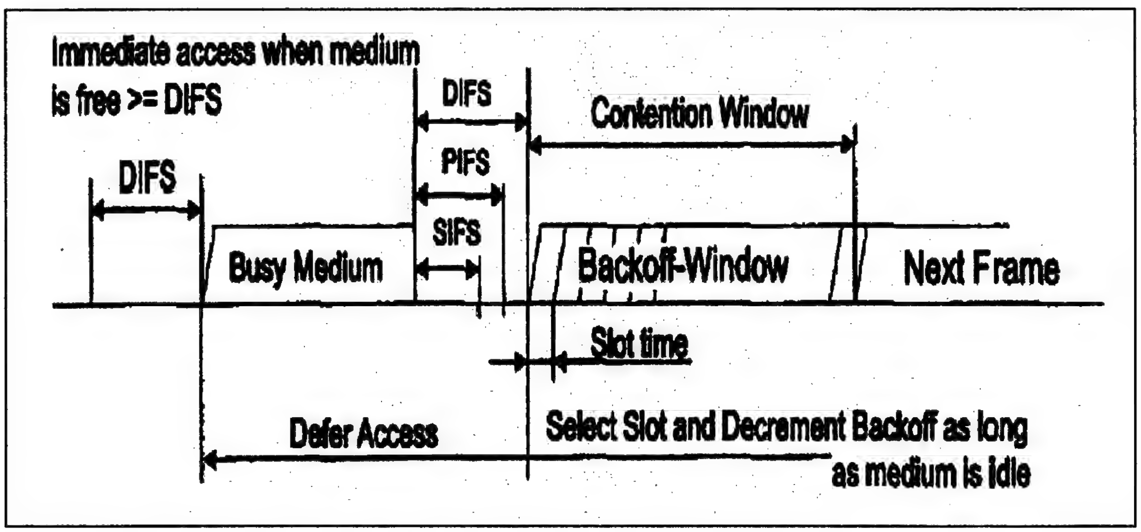

Figure 3. Timing Relationships in the 802.11 Standards (From Ref. [2]).

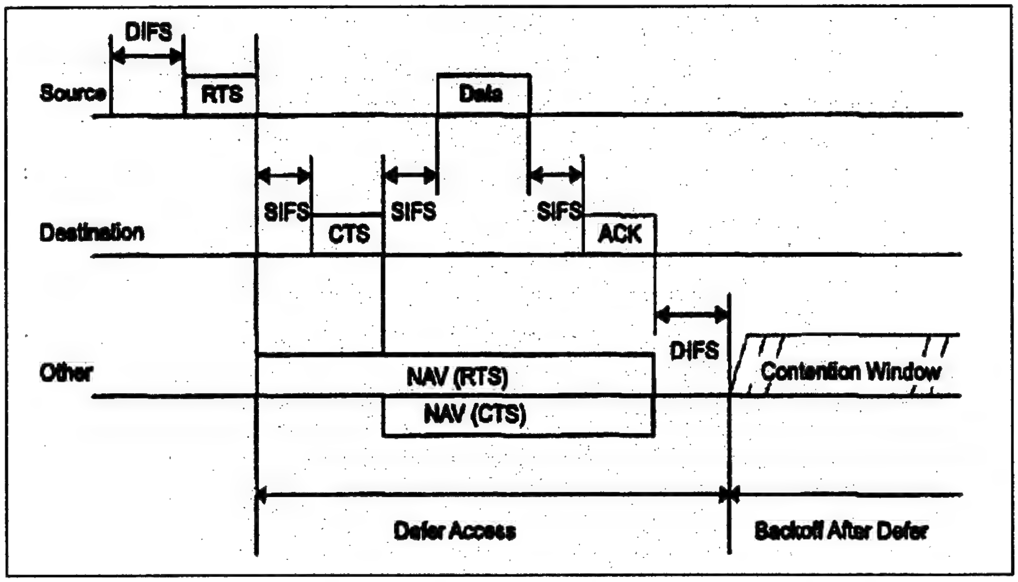

Figure 4. Timing in a Data Transmission Scenario (From Ref. [2]).

To allow for the proper operation of a WLAN BSS the 802.11 MAC delineates the exchange of three basic frame types: data frames, control frames, and management frames. Data frames are used to convey user data between the WLAN nodes, management frames are used to allocate and report on network resources (e.g. authentication/deauthentication, association/disassociation, probing, and beaconing), and control frames are utilized to control access to the wireless medium (e.g. RTS, CTS, and ACK frames). The medium access process and data delivery are at the core of the model presented here, therefore management frames and their roles will not be addressed.

Each 802.11 MAC layer frame, or MAC protocol data unit (MPDU), consists of a MAC header, a frame body, and a MAC trailer, which is essentially the frame check sequence (FCS) used in detecting bit errors in the frame. The basic frame format is shown in Figure 5. The contents of the "Frame Body" and "Frame Control" fields differentiate data, control, and management frames. Data frames will obviously have user data in the "Frame Body" field and are therefore variable in length. ACK, RTS, and CTS frames have specifically delineated fields in the frame body and are of constant length. It is important to note that the frame format and transmission speed within the BSS are closely related. Per the specification, data frames may be sent at any of the rates supported by the standard, while control frames must be transmitted at one of the mandatory data rates to ensure seamless communication between possibly disparate 802.11a implementations.

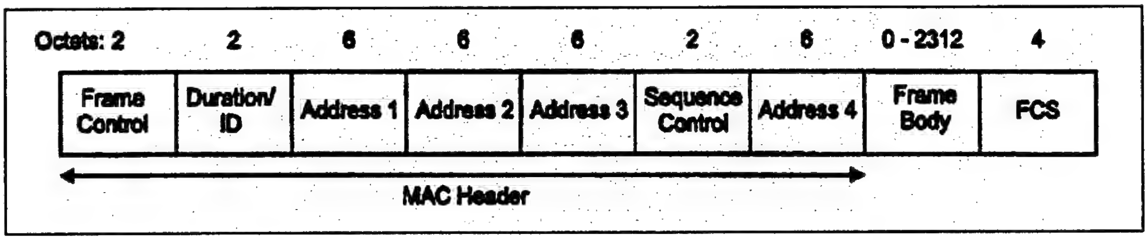

Figure 5. The 802.11a MPDU (From Ref. [2]).

In the preceding paragraphs the key elements of the 802.11a MAC layer have been summarized, and each will play a role in the description of the 802.11a model discussed in the remainder of this thesis. The data rates supported by the 802.11a standard are clearly much higher than those of the original 802.11 specification and the 802.11b addendum. But, the 802.11 MAC was designed to be independent of the PHY layer so with some minor differences, the 802.11a mechanisms for access to the RF medium are essentially the same as in prior implementations. In short, the consumer is benefiting from a nearly five-fold increase in data rate with very little change in the MAC layer.

C. THE 802.11A PHYSICAL LAYER

The IEEE 802.11a standard calls for the use of COFDM in the PHY layer to realize the full 6-54 Mbps range of data rates. OFDM is a multicarrier communications scheme in which a single high-rate data stream is split into lower-rate data streams that are subsequently transmitted in parallel over a number of subcarriers. The subcarriers overlap and the inter-carrier spacing are chosen such that all the subcarriers are orthogonal to each other. OFDM is not a new technology; it has been used in digital audio broadcasting (DAB) and digital video broadcasting (DVB) since the 1970's [6]. However, it has only recently been adopted for use in high data rate wireless packet-based communications. OFDM was selected for use in the 802.11a standard based on its mitigation of many of the difficulties associated with wireless communications in the 5 GHz band such as multipath fading and transmission power level restrictions [17]. The 802.11 and 802.11b standards utilize spread spectrum communications in the PHY layer, but spread spectrum encoding at 5 GHz with low power levels would not provide the requisite operational range in office, campus or home environments (due to the inverse proportionality of frequency and distance). OFDM offers high-rate data transmission with a minimal increase in the complexity of the PHY layer implementation.

The 802.11a standard specifies a channel spacing of 20 MHz with a 16.56 MHz 3-dB transmission bandwidth per channel. The specified channelization for 802.11a (in the

United States) in the 5 GHz UNII bands is shown in Table 4. Each UNII band has a corresponding maximum output power level per FCC regulations (see Table 5) suggesting use of the lower band in shorter-range applications (i.e., the home WLAN market), use of the middle band in office-like environments, and use of the upper band in longer-range applications (i.e. cross-campus WLAN bridging and warehouse settings). The power levels are given assuming full use of the allocated bandwidth along with the levels in mW/MHz if only a portion of the bandwidth is used. Within a channel, each OFDM transmission consists of 52 separate subcarriers, 48 of which are used to transmit data while the other four are used as pilot signals for hardware synchronicity. Each subcarrier is spaced 312.5 kHz from adjacent subcarriers and each is modulated independently. Binary phase shift keying (BPSK), quadrature phase shift keying (QPSK), and both 16- and 64-quadrature amplitude modulation (QAM) can each be used to modulate the subcarriers in conjunction with specific COFDM coding rates to achieve each of the supported 802.11a data rates.

Band (GHz)	**Channel Number**	**Center Frequency (MHz)**
UNII lower band (5.15 – 5.25)	36	5180
	40	5200
	44	5220
	48	5240
UNII middle band (5.25 – 5.35)	52	5260
	56	5280
	60	5300
	64	5320
UNII upper band (5.725 – 5.825)	149	5745
	153	5765
	157	5785
	161	5805

Table 4. Channelization in the 802.11a Standard (From Ref. [4]).

Frequency Band (GHz)	**Maximum Output Power (mW) [with up to 6 dBi antenna gain]**
5.15 – 5.25	40 (or 2.5 mW/MHz)
5.25 – 5.35	200 (or 12.5 mW/MHz)
5.725 – 5.825	800 (or 50 mW/MHz)

Table 5. 802.11a Maximum Output Power Levels (From Ref. [4]).

The IEEE 802.11a standard specifies the use of COFDM with convolutional forward-error correction (FEC) coding. FEC coding allows for the correction of errors found in the weakest subcarriers that are adversely affected in the multipath fading channels characteristic of a wireless communications link. 802.11a specifies the use of several coding rates in conjunction with the modulation schemes listed above: 1/2, 2/3, and 3/4. The modulation scheme and coding rate combinations are shown in Table 6 along with their corresponding data rates. The shift register size, or constraint length, for the convolutional coding computations in the 802.11a standard is set at seven [4].

Data Rate (Mbps)	Subcarrier modulation	Coding Rate (R)	Coded bits per subcarrier	Coded bits per OFDM symbol	Data bits per OFDM symbol
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

Table 6. Coding and Modulation in the 802.11a Standard (From Ref. [4]).

COFDM also serves to mitigate another adverse feature of wireless multipath fading channels: intersymbol interference (ISI) caused by the multipath delay spread. 802.11a utilizes a high symbol rate (250 kilosymbols per second) to achieve higher data rates, therefore a high degree of ISI due to multipath delays could obviously impact performance. Typical maximum multipath delay spreads in a WLAN environment range

In summary, COFDM was chosen as the 802.11a PHY layer technology based on its ability to counter the negative effects of low power, high data rate wireless packet transmission in a multipath fading environment. The use of orthogonal subcarriers allows utilization of the allotted bandwidth through conventional modulation techniques when applied with convolutional FEC coding. ISI is greatly reduced through the use of an 800 ns guard time prefix while ICI is minimized by using a cyclic extension of the OFDM symbol during that guard interval. The MAC and PHY layer characteristics introduced in this chapter will be applied in Chapter III where the 802.11a baseline model is detailed. The timing and medium contention schemes introduced here are features of the model as are the PHY layer dependent characteristics, like the SIFS and Slot time. The PHY layer is modeled using data rate-dependent COFDM channels within the framework of the OPNET transmission scheme. The 802.11a baseline model and the OPNET modeling and simulation tool are both discussed in Chapter III.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IEEE 802.11A PROTOCOL MODEL

The IEEE 802.11a WLAN model presented here was constructed using the OPNET network modeling and simulation tool. The model includes both the MAC and PHY layers and incorporates features adapted from the 802.11a OPNET model created by Dr. Sunghyun Choi of Philips Reasearch Labs [1]. OPNET was chosen as the software environment for this model based on its ability to replicate the preponderance of 802.11a features, excepting some PHY layer characteristics, with a high degree of design control. OPNET is geared more toward exploring network-wide design issues and conducting research at the MAC layer and above (i.e. IP, TCP, UDP) than for examining the physical behavior of wireless links. This model does, however, incorporate the essential 802.11a PHY characteristics. Other modeling and simulation tools used for WLAN research include MATLAB and NS2.

A. OPNET AND THE 802.11 STANDARD MODEL

The IEEE 802.11a baseline model was created using OPNET Version 7.0B with software patch level 11 on a Windows NT 4.0 platform. The OPNET simulation tool is capable of modeling the majority of modern networking protocols and standards. Within the OPNET interface environment networks are modeled in a layered fashion, not unlike the actual protocols themselves. The highest level of the modeling framework is the *network domain*, where the overall topology of the network is defined [18]. The network components (e.g., STA, AP, server, router) are referred to as *node models*, each of which is further subdivided into *node objects*. These node objects represent the functions that take place within a given node model (e.g., MAC, TCP, IP encapsulation). A node object typically consists of a *process model*, or *state transition diagram* (STD). When a state is entered or exited the underlying model functions and OPNET-specific functions (called *kernel procedures*) of a node object are called from the *enter executives* or *exit executives* of the state. These executives essentially dictate the operation of the STD and are written in the OPNET-specific Proto-C language, as are the underlying functions and kernel procedures. Figure 7 depicts each of the network domain levels. Most, but not all node objects consist of process models. For example, the WLAN transmitters and receivers used here can only be in a single state: either transmitting or receiving, respectively. The attributes of these node objects are therefore specified via a graphical user interface.

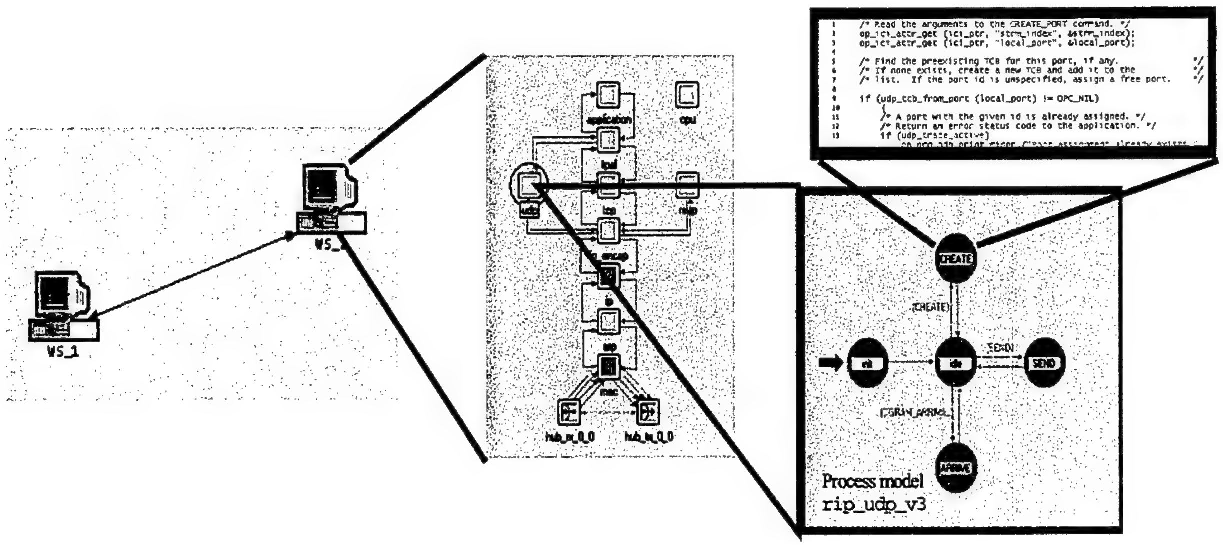

Figure 7. OPNET Design Environment (From Ref. [19]).

The simulation of frame transmission between wireless node models in the OPNET network domain, based on the attributes of the transmitter and receiver, is accomplished by way of a *transmission pipeline*. The pipeline is broken down into a series of *pipeline stages*, each of which takes the form of a C++ executable file that is designed to emulate a transmission-specific task. The pairing of a wireless transmitter with a receiver, the calculation of the path loss, and the determination of the link bit error rate (BER) are all examples of pipeline stage functions. These stages are invoked when a frame is transmitted by a WLAN transmitter on a channel specified by the MAC. Unlike actual WLAN systems, a single OPNET channel is associated with a particular data rate. The transmission/reception channel is identified to the transmitter/receiver from the MAC by a series of packet streams. In other words, the passing of a frame from the MAC layer to the PHY layer is modeled by a series of separate data rate-dependent packet streams. *Packet* is a generic term used to refer to the vehicle for data transmission within an OPNET model. The OPNET software package contains a variety of packet formats. The actual WLAN frames in the 802.11a model are emulated by a set of packet formats. OPNET allows for customized packet formats that include fields that accurately represent an actual frame format as well as "null" data fields. These "null" packet fields are understood by OPNET to not contain any actual simulated data, but rather can be used by the simulation kernel to pass information between the MAC process model and the transmission pipeline stages.

OPNET's software package includes a number of standard models that represent common networking protocols, one of which is a basic 802.11 WLAN model that incorporates some features of the 802.11b addendum. The OPNET standard model package does not include a model of the 802.11a protocol because the specification was only recently approved. The 802.11 WLAN model standard model served as the foundation for the 802.11a model along with a number of features adapted from the Philips Reasearch Labs 802.11a model [1]. The 802.11 standard model includes node models of wireless STAs (fixed and mobile), APs, and servers. Each node model includes a MAC process model (called *wlan_mac*), a transmitter object and a receiver object. These three objects, accompanied by the wireless pipeline stages, comprise the MAC and PHY layers. The *wlan_mac* process model is the heart of the 802.11 MAC model, as it contains all of the Proto-C code and functions representative of the rules that govern the possible states in the medium contention process. The 802.11 *wlan_mac* supports the 1, 2, 5.5 and 11 Mbps data rates and incorporates the other features of the 802.11 MAC discussed in Chapter II with the exception of the optional PCF polling scheme.

The 802.11 standard model PHY layer is represented by the transmitter object, receiver object, and the wireless pipeline stages. There are a total of 14 wireless pipeline stages, four of which were developed specifically for the 802.11 WLAN model. Those four stages concern the determination of eligible WLAN receivers for a BSS, the matching of receiver and transmitter channels, and the calculation of the propagation delay and received signal power. Each WLAN transmitter/receiver has four packet streams from/to the *wlan_mac* process model, one for each 802.11b-supported data rate.

The 802.11 standard model proved to be a good foundation for the 802.11a model when used in conjunction with the MAC layer features of the Philips Research Labs model. The PHY layer required the greatest number of alterations to develop a comprehensive representation of the 802.11a standard, as will be seen in the following sections where the 802.11a model is presented in detail.

B. THE 802.11A BASELINE MODEL

The IEEE 802.11a model was constructed by altering the OPNET 802.11 standard WLAN model and incorporating a number of features from the Philips Research Labs 802.11a OPNET model. The architecture modeled is that of an infrastructure BSS with a single fixed AP and a variable number of mobile STAs. The infrastructure BSS was

chosen vice the independent BSS as it will likely be the configuration of choice in home, office, or campus environments. The 802.11a MAC and PHY layers are identical in both the AP and the STA node models with the exception of several user defined attributes that will be covered in greater detail later. Layers above the MAC in the node models are somewhat different, since the AP has to interface with a wired external network. Figure 8 depicts an 802.11a WLAN model with a single AP and ten mobile STAs in the OPNET network domain.

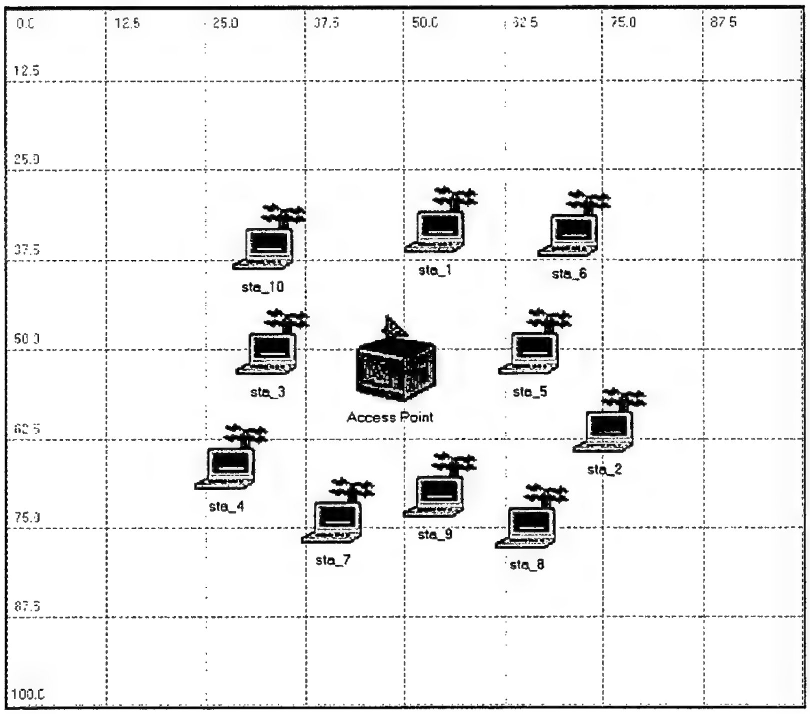

Figure 8. An 802.11a WLAN in OPNET.

Each 802.11a STA node model is called *wlan_wkstn_adv_11a* while the AP node model is called *wlan_ethernet_router_adv_11a.* These two node models are used in tandem to create an 802.11a infrastructure WLAN within the OPNET design environment. The node object composition of *wlan_wkstn_adv_11a* is shown in Figure 9 while that of the *wlan_ethernet_router_adv_11a* is depicted in Figure 10. Both contain higher layer node objects representing common protocols like *tcp*, *udp*, *ip*, *ip_encap*, and *arp*. The *wlan_ethernet_router_adv_11a* has two interfaces, one for a wired Ethernet network and another for the WLAN. The *wlan_mac_11a* node object represents the 802.11a WLAN MAC in each node model, while the *wlan_port_tx* and *wlan_port_rx* objects represent the components of the WLAN transceiver. The *wlan_mac_11a*,

wlan_port_tx, and *wlan_port_rx* node objects and their interactions are the heart of the OPNET 802.11a baseline model.

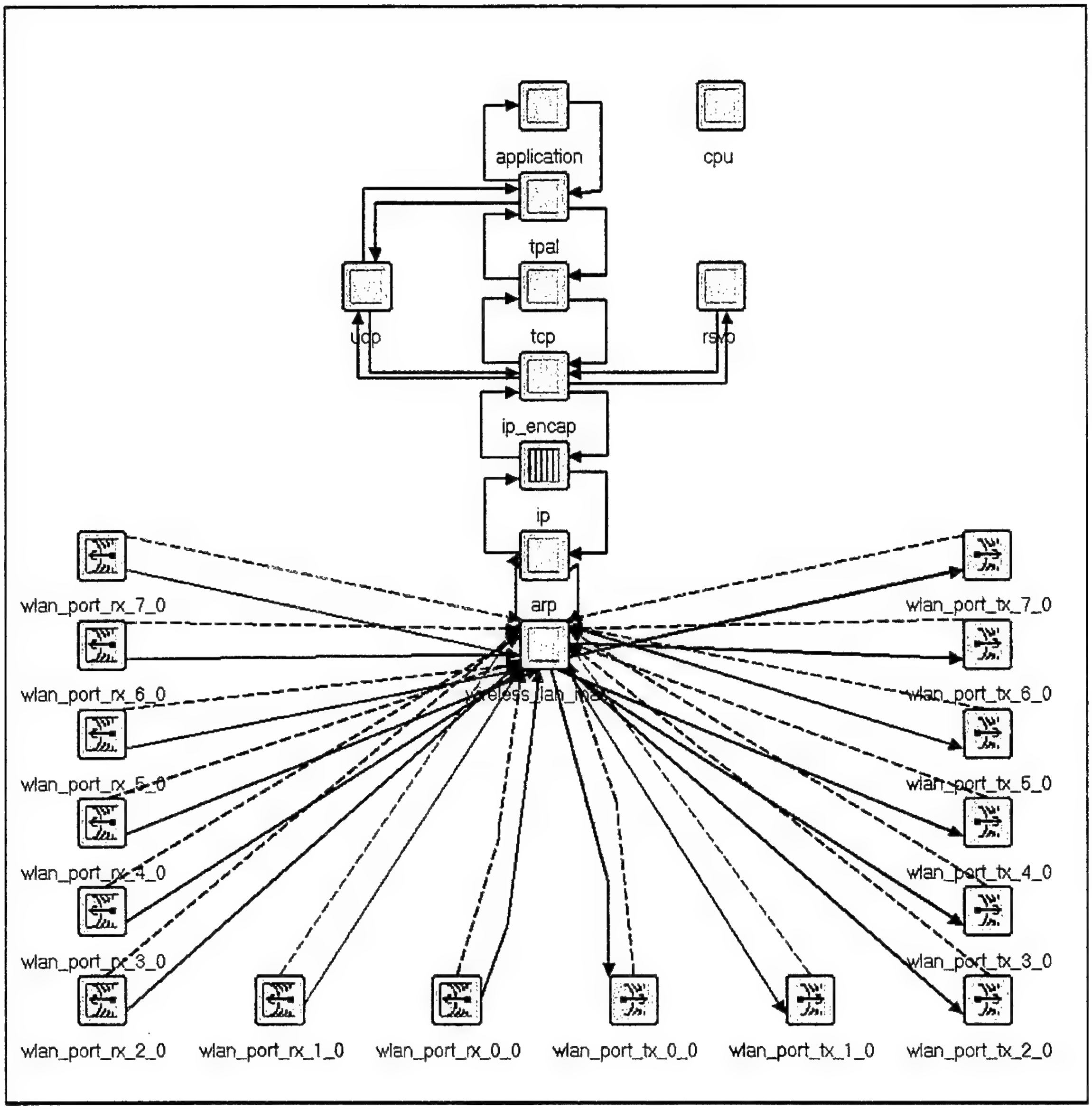

Figure 9. The 802.11a Model STA Node Object.

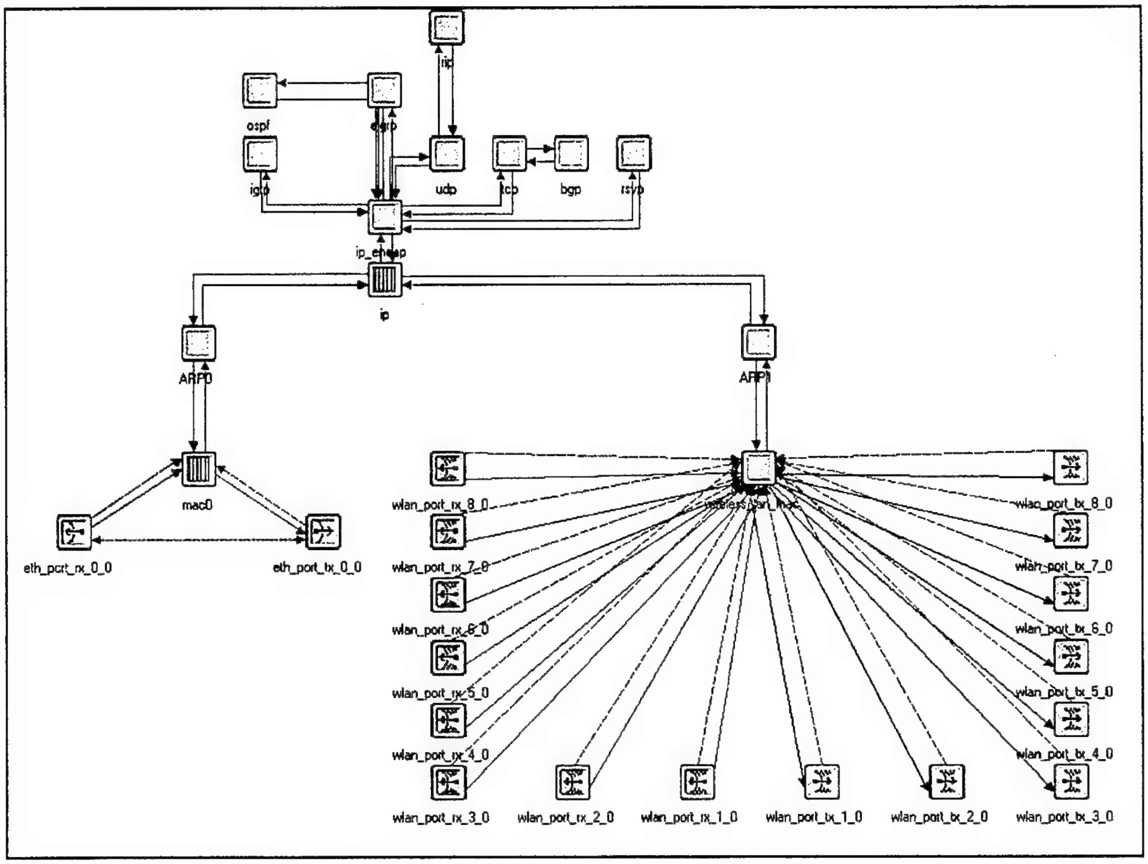

Figure 10. The 802.11a Model AP Node Object.

1. The 802.11a Model MAC Layer

The *wlan_mac_11a* node model and its accompanying process model, adapted from the 802.11 *wlan_mac* process model, are used to represent the 802.11a MAC. The states and transitions of the *wlan_mac_11a* process model are the same as those of *wlan_mac*, with the 802.11a functionality realized through modifications to the underlying Proto-C code and function calls. The *wlan_mac_11a* process model is shown in Figure 11. Two new functions were added and changes were made to four of the 13 functions already defined in the model code. The Proto-C code make-up of *wlan_mac_11a* is provided in its entirety in Appendix A.

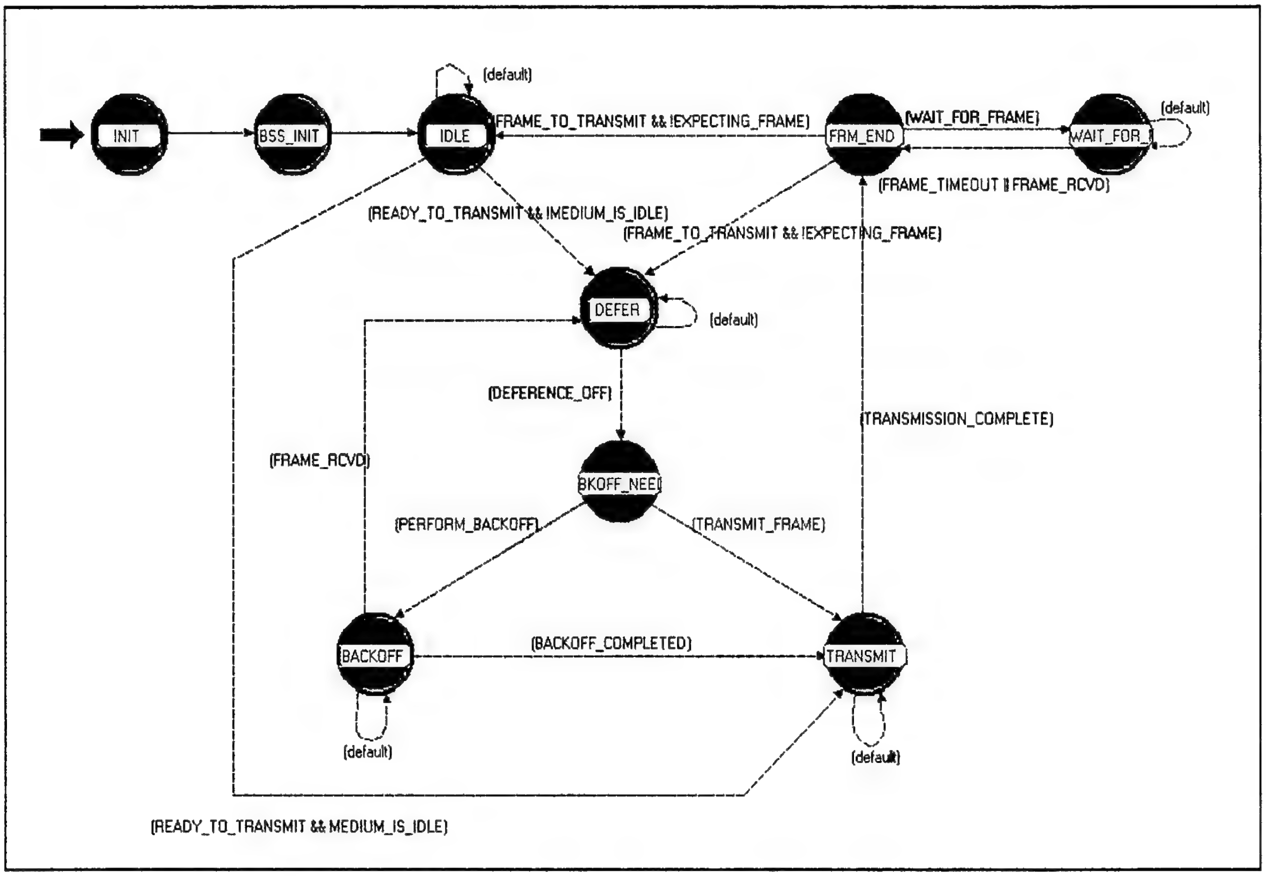

Figure 11. The *wlan_mac_11a* Process Model.

The behavior of *wlan_mac_11a* is governed by a number of user-defined parameters, lumped under the *Wireless LAN Parameters* attribute and selected via an OPNET graphical user interface. The critical parameters are listed in Table 7. Some parameters may be assigned any numerical value, but the values shown are those specified by the 802.11a standard and these must be selected for the model to function correctly. The OFDM *Physical Characteristics* attribute ensures that the correct values for the SIFS time, slot time, and minimum and maximum contention window size (as specified in Chapter II) are defined when the simulation begins. It also provides for the definition of the 802.11a PLCP preamble and PLCP header transmission durations, which are used by the stations to correctly set their NAVs when RTS/CTS is enabled by selection of a non-zero RTS threshold value. The RTS Threshold can take any value up to 2347. The short and long retry limits delineate the number of times a STA may attempt to retransmit frames that are shorter or longer, respectively, than the RTS Threshold value. The *AP functionality* parameter lets the user identify the AP in a BSS if the WLAN is an infrastructure WLAN.

Parameter Name	Values
RTS Threshold	Any integer < 2347
Fragmentation Threshold	Any integer < 2347
Short Retry Limit	4
Long Retry Limit	7
Data Rate	6, 9, 12, 18, 24, 36, 48, or 54 Mbps
Physical Characteristics	OFDM
AP Functionality	Yes or No (Boolean)

Table 7. User-defined *Wireless LAN* Parameters.

The *data rate* attribute is provided for the user to select the maximum operational rate for the exchange of data frames within the WLAN by a given STA. Recall from Chapter II, however, that the control frame transmission rate must be one of the three rates from the mandatory rate set. To determine the correct control frame rate, a function was added to the wlan_mac_11a process model to select the highest possible control frame speed given the data frame transmission rate. This function was adapted from the Philips Research Labs model [1]. In addition, the STA may receive frames from another STA that might not be operating at the same data transmission rate. The receiving STA must then determine the speed at which to respond with either a CTS or ACK frame based on the incoming frame type. The capability to deal with this scenario was added to the *wlan_mac_11a* process model Proto-C code using a mechanism similar to the function described above.

Once the transmission data rate of a given frame has been determined, the frame must then be passed to the PHY layer for transmission. The four packet streams connecting the *wlan_mac_11a* node object and the PHY layer (i.e., the transmitter and receiver node objects) in the STA and AP node models of the 802.11 standard model were replaced with eight packet streams representing each of the 802.11a possible data rates. These streams can clearly be seen in Figures 9 and 10. Each stream has an accompanying statistic wire (the dashed lines in the figures) to emulate the physical carrier sensing capability of the STA. These statistic wires inform the *wlan_mac_11a*

process model when either a transmitter or receiver is busy. The use of transmitter and receiver node objects to model the PHY layer will be covered in the next subsection.

The virtual carrier sensing capability of the 802.11 standard model MAC had to be altered to account for both the format and length of the OFDM PPDU and the PLCP preamble and header transmission durations. Accordingly, each NAV duration calculation in the underlying functions of *wlan_mac_11a* was modified to emulate the correct timing relationships. The new NAV durations were calculated in two steps. First, a new function was added to determine the duration of the body of a given PPDU, to include the PLCP Service Data Unit (PSDU), the SERVICE field, the tail bits, and the required padding to complete the OFDM symbol. This function was adapted from the Philips Research Labs model [1]. Secondly, the PLCP header duration and the PLCP preamble duration were added to completely emulate the overhead associated with the transmission of an OFDM PPDU. Note that the PLCP header and PLCP preamble durations are the same for each frame regardless of its format since these two packet fields are always transmitted at the lowest data rate in the mandatory rate set. An identical change was made to the exit executives of the *FRM_END* state to accurately model the operation of the timer used when waiting for expected response frames from other STAs.

Two proto-C code error repairs to the 802.11 standard model were also adapted from the Philips Research Labs' 802.11a model to ensure proper operation of the *wlan_mac_11a* process model [1]. The first corrects the calculation of the EIFS time while the second corrects the erroneous calculation of the remaining length of a data frame during the frame fragmentation process. Also adapted from the Philips Labs' model is the ability to track the net amount of MAC layer traffic sent or received by a given station for analysis following a simulation. The overhead associated with the PHY layer can therefore be disregarded if the goal is to just analyze the amount of MAC layer traffic handled by the WLAN. This feature was included in the model but is not demonstrated in this thesis. The size of the MSDU is passed between STAs using a null field in the OPNET formatted packet. The two packet formats associated with the 802.11a model are the *wlan_data_802_11a* and *wlan_control_802_11a* packets, shown in Figures 12 and 13 with their field names and bit sizes. The "MPDU size" field is used to pass the size of the MPDU between STAs for use in simulation data analysis. These two 802.11a packets are identical to the 802.11 standard model packet formats with the exception of the "MPDU size" field and the "Rate" field. The "Rate" field is used in the

PHY layer and will be described in greater detail in the next subsection, where the model's PHY layer components are presented.

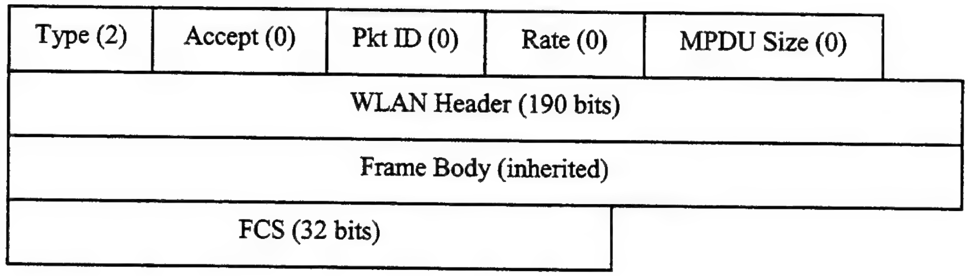

Figure 12. The *wlan_data_802_11a* Packet Format

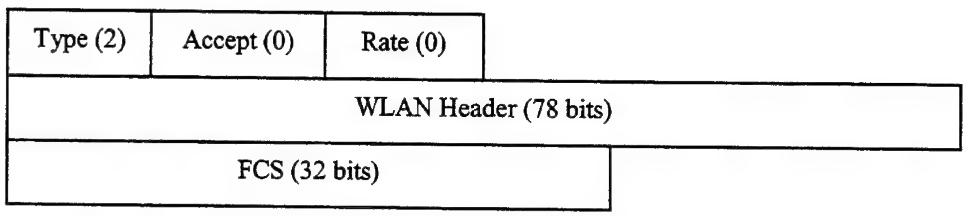

Figure 13. The *wlan_control_802_11a* Packet Format

2. The 802.11a Model PHY Layer

The PHY layer of the 802.11a model is represented by eight *wlan_port_tx* and eight *wlan_port_rx* node objects in conjunction with the 14 wireless transmission pipeline stages. The eight transmitters and eight receivers emulate the operation of a single WLAN transceiver. A single transceiver is modeled in this manner because each OPNET transmitter and receiver node object is wedded to a specific modulation scheme. To realize the eight specified data rates with their accompanying modulation and coding rate combinations a total of 16 transmitter and receiver objects are required. When a packet is sent from the MAC to the PHY layer, it will travel on one of the eight packet streams to the appropriate transmitter node object associated with the specified data rate. When the packet is sent through the pipeline stages, only those receivers associated with that particular data rate may receive the packet. This model design allows for the emulation of transmission between STAs and APs at specifically designated data rates. The transmitter and receiver node objects are modeled as isotropic antennas with typical isotropic transmission and reception patterns and unity gains.

Like the MAC layer, the behavior of the PHY layer is partially governed through the use of user-defined parameters that are attributes of the receiver and transmitter nodes. These parameters fall under the heading of four attributes: *modulation*, *channel*, *noise figure*, and *ecc threshold*. The *channel* attribute is used to further specify the parameters associated with each OPNET wireless transmission channel (i.e., each data rate), while the *modulation* attribute is used to specify the transmitter and receiver's modulation schemes. The *noise figure* attribute allows for the selection of the receiver noise figure while the *ecc threshold* specifies the acceptable BER upper bound for received packets. The parameters of each attribute and their nominal values are shown in Table 8.

Attribute	Parameter	Values
Channel	Supported Packet Formats	*wlan_data_802_11a* and *wlan_control_802_11a*
	Bandwidth	16,560 kHz
	Base Frequency	One of: 5171.7, 5191.7, 5211.7, 5231.7, 5251.7, 5271.7, 5291.7, 5311.7, 5736.7, 5756.7, 5776.7, 5796.7 MHz
	Spreading Code	Disabled
	Processing Gain *	0.0
	Power **	0.04, 0.2 or 0.8 W
Modulation	Modulation Scheme	One of: Ofdm_6Mbps Ofdm_9Mbps Ofdm_12Mbps Ofdm_18Mbps Ofdm_24Mbps Ofdm_36Mbps Ofdm_48Mbps Ofdm_54Mbps
Noise Figure*	N/A	Any number > 1.0 (nominally ~ 5)
ECC Threshold*	N/A	Any number (nominally ~ 1×10^{-5})

* Receiver Only
** Transmitter Only

Table 8. Attributes of the *wlan_port_tx* and *wlan_port_rx* Node Objects.

The first parameter of the *channel* attribute is the supported OPNET packet formats. These are specified so that named packet fields can be accessed and/or modified as a packet traverses the pipeline stages. The *wlan_data_802_11a* and *wlan_control_802_11a* packet formats presented in the previous subsection are specified for use here. The channel bandwidth delineated in the 802.11a standard is used as the bandwidth value, while any of the base frequencies of the channels listed in Table 4 are acceptable as the base frequency value. Note that the corresponding per band transmission power level as specified in Table 5 must be used in conjunction with the selected channel frequency. For example, if a base frequency of 5736.7 MHz is chosen then the transmitter power parameter must be set at or below 0.8 W. The spreading code parameter is applicable only for 802.11 or 802.11b WLANs and therefore is disabled in the 802.11a model. By the same token, the processing gain is an additive gain associated with direct sequence spread spectrum communications and accordingly should be set to zero here.

The modulation attribute setting plays a key role in emulating PHY layer channel characteristics. The *dra_ber* pipeline stage uses the transmitter and receiver modulation attribute to determine the BER of the packet transmission by way of a modulation table look-up based on the link SNR calculated in the *dra_snr* pipeline stage. A modulation table contains a range of BER versus E_b/N_o values, and when the table look-up kernel procedure is invoked in the pipeline stage the BER is determined based on the previously computed SNR. In other words, the BER is a function of the channel modulation scheme. Each subcarrier of an OFDM transmission is modulated according to the scheme outlined in Table 5, but the OPNET simulation environment is not detailed enough to support the emulation of each individual subcarrier. Instead, the 802.11a model is designed such that a single OFDM transmission is treated by OPNET as an aggregated signal based on the data rate of the transmission.

Recall that each 802.11a data rate is associated with a specific subcarrier modulation type and convolutional coding rate. Eight new modulation tables were created in OPNET to represent the BER versus E_b/N_o characteristics of an OFDM transmission at each data rate. The modulation tables were created using values taken from BER versus E_b/N_o curves found in reference [17] and shown in Figure 14. These curves represent values associated with OFDM transmissions in additive white gaussian noise (AWGN) for a constraint length seven convolutional code given the subcarrier modulations and coding rates for each 802.11a data rate. Although these curves fail to capture the Rayleigh fading behavior of a typical wireless communications link, they

represent the best, most current data available for use in emulating the actual 802.11a PHY layer. Ongoing detailed simulations and measurements of 802.11a PHY layer transmission characteristics using other tools may soon provide more accurate data for incorporation in future versions of this model [20].

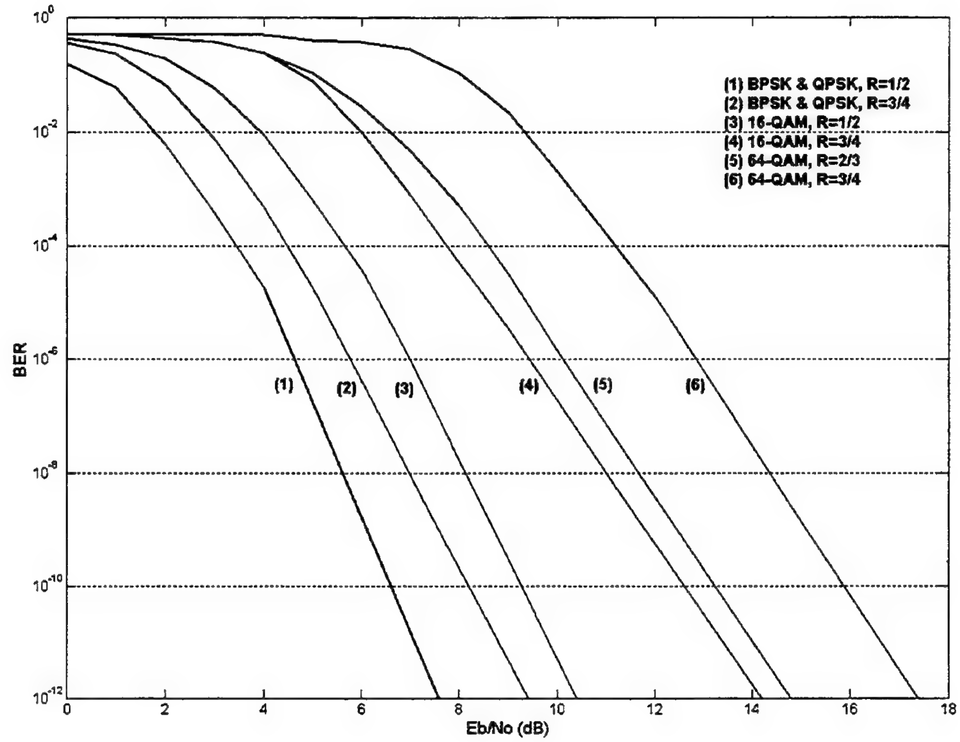

Figure 14. BER versus E_b/N_o Curves for OFDM in AWGN.

The *modulation* attribute specified by the user represents the modulation table associated with the maximum rate of data traffic transmission within the WLAN. The selected data rate is not static however, and changes based on the frame type (i.e., data or control) and possibly with the transmission rate of incoming data packets. When a packet is sent to the PHY layer by the MAC layer for transmission at a specified data rate, the packet is sent to the transmitter object possessing the modulation attribute that corresponds to that data rate. In this fashion the most accurate BER approximation is assigned to the calculated SNR in the pipeline stages. The data rate must also be passed to another pipeline stage, *dra_txdel_11a*, for use in calculating the transmission delay associated with that data rate. The *dra_txdel_11a* stage must therefore be able to track the rate at which a frame is sent and adjust accordingly. It does so through the use of the

"Rate" field, a customized packet field added to both the *wlan_data_802_11a* and *wlan_control_802_11a* packet formats. Just prior to a packet's release from the MAC layer to the PHY layer, the data rate at which that frame is being transmitted is stored in the packet's "Rate" field. When the *dra_txdel_11a* pipeline stage is invoked, the contents of that field are accessed and the proper data rate is used in calculating the transmission delay. The *dra_txdel_11a* pipeline stage code is provided in Appendix E.

The 802.11 standard model uses a default value of 1.0 for the *noise figure* (f_n), which equates to a 0 dB thermal noise contribution at the receiver. A 0 dB thermal noise value represents the ideal-case reception of a frame at the receiver where the thermal noise value is negligible. Rather than using this default value a nominal value of 5.01 was selected based on the reported noise figure of the Lucent WaveLAN™ 802.11b network interface card, a popular WLAN implementation [21]. The selection of a noise figure value found in a fielded WLAN helps reduce the artificiality of the OPNET transmission process. The total background noise (B_N) that effects the received packet (as calculated by the *dra_bkgnoise* pipeline stage) is therefore given by:

$$B_N = BW(290 f_n)k + A_N + I_N$$

where Boltzmann's Constant $k = 1.379\times10^{-23}\ {}^{J}\!/\!_{K}$, 290 is the receiver background temperature in degrees Kelvin, BW is the transmission bandwidth of 16.56 MHz, the OPNET default ambient noise A_N = 1x10^{-26}, and the inter-packet interference I_N is as calculated in the *dra_inoise* pipeline stage. Inter-packet interference results from the occasional slight overlap of two packets as one completes the reception process while the other is just arriving at the receiver. The overlap is so small as to not be considered a collision but rather a source of noise. The value of I_N is rarely non-zero, highly, and extremely small.

To add even greater fidelity to the losses encountered in the transmission pipeline the *wlan_power* pipeline stage's default free space path loss calculation was altered. The path loss equation was modified to more accurately reflect the losses that might take place in a typical office-like environment. The path loss (P_L) as computed in *wlan_power* is given by:

$$P_L = \frac{\lambda^2}{16\pi^2 d^n}$$

where λ denotes the wavelength associated with the channel's center frequency, d is the distance that separates the transmitting and receiving STAs, and n is the path loss exponent. A value of n=2 corresponds to the OPNET default of a simple free space path loss assignment whereby the only losses result from the attenuation of the signal through the air in a straight line from the transmitter to the receiver. However, in a typical indoor office environment the signal would suffer from increased attenuation as it passes through partitions, walls, doors, floors and ceilings. A number of studies have empirically determined typical path loss exponent values in office, school, and residential environments [22, 23, 24]. In particular, Medbo and Berg obtained a value of n=3.8 when assessing the losses between rooms in a typical school setting [24]. This value of n was selected for use in the 802.11a baseline model as it falls within the range of exponents found in the other studies and because it was obtained in an environment that more closely resembles the settings we are interested in.

The *ecc threshold* attribute allows the user to specify an upper bound for the acceptable BER of a received packet. If the BER exceeds the specified threshold then the packet is marked as unacceptable in the final pipeline stage and is then discarded by the *wlan_mac_11a* process model. This procedure is used to emulate the WLAN's limited ability to detect and correct frame errors. Typical 802.11 and 802.11b implementations are able to cope with BERs up to around 1×10^{-5} [25].

The 802.11a model presented here offers a new approach to comprehensively modeling both the MAC layer and PHY layer attributes of a wireless protocol using the OPNET modeling tool. OPNET modeling efforts have traditionally focused on the MAC layer and above at the expense of PHY layer features and their effects. The baseline model outlined above serves as a starting point for further research involving the 802.11a protocol and its MAC and PHY layer characteristics.

C. 802.11A BASELINE MODEL SIMULATION RESULTS

The 802.11a baseline model was used in an OPNET simulation to test and verify its performance. The goal of the simulation was to confirm proper operation of the model vice the analysis of a particular aspect of the protocol's behavior or examining a specific network performance characteristic. The simulation was conducted using a variation of the OPNET 802.11 standard model's *wlan_deployment* scenario. In this scenario the behavior of a single infrastructure 802.11a WLAN was examined within the framework of a deployed wide area network (WAN) to better emulate the configuration of an actual

network. The WLAN is connected to an IP gateway (i.e., an enterprise router) which is in turn connected to an IP cloud used to represent the backbone Internet. The network's traffic servers are located on the other side of this IP cloud via a firewall. These servers are used as the source and destination of the file transfer protocol (FTP) and hypertext transfer protocol (HTTP) traffic that is exchanged with the STAs in the 802.11a WLAN during the simulation. The high-level network environment is depicted in Figure 15. The red octagon in Figure 15 titled *site_1* represents the 802.11a WLAN BSS subnetwork. Within that subnetwork are the STAs and the AP that comprise the WLAN, as seen in Figure 16.

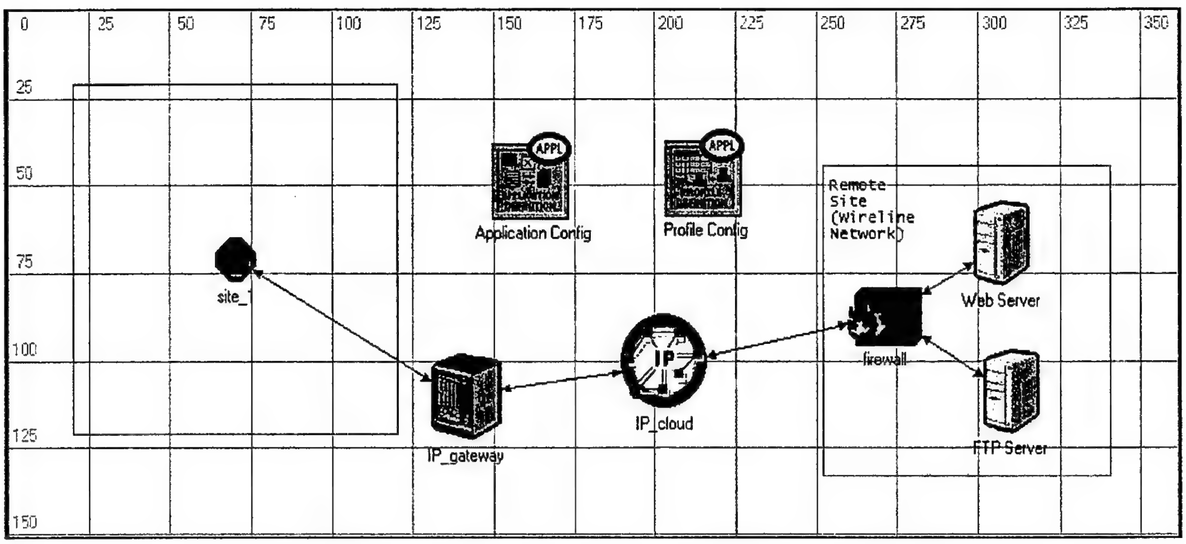

Figure 15. Simulated 802.11a Network Environment.

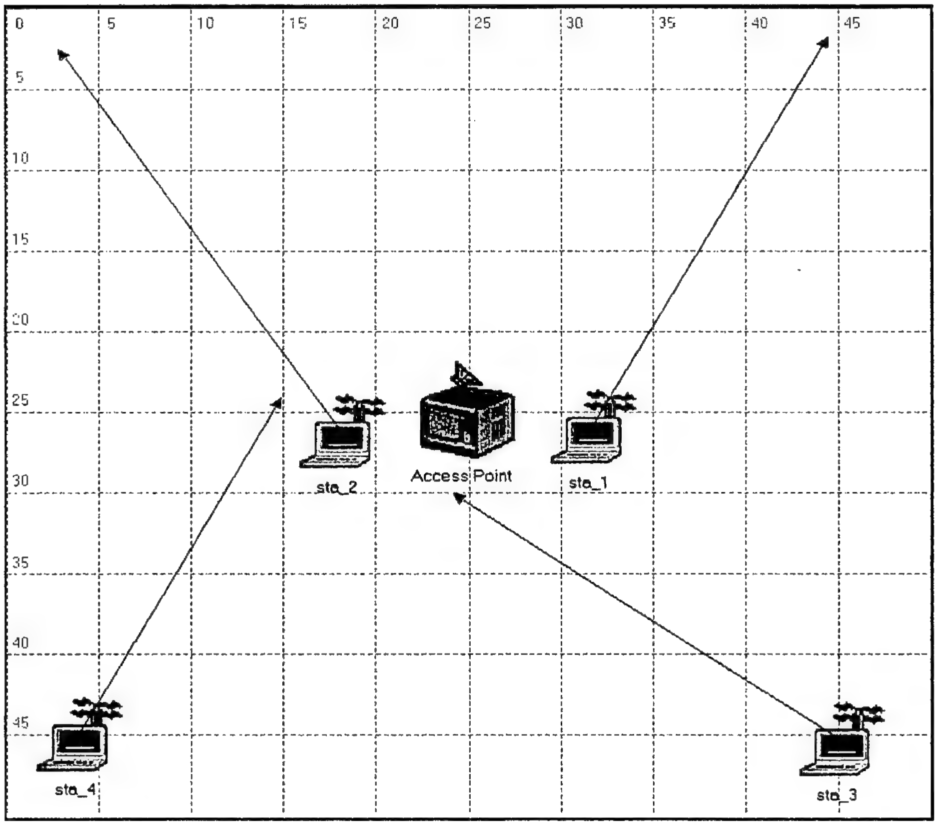

Figure 16. The Simulated 802.11a WLAN BSS.

A single fixed AP and four mobile STAs were chosen as the WLAN configuration for the simulation. This small WLAN was selected both to limit the scope of the simulation and to achieve reasonable simulation durations. The simulation of this small WLAN took approximately two hours given the simulation parameters outlined below. The four arrows in Figure 16 represent the path of each station as the simulation progresses, with STA distances from the AP varying. In general, two STAs are closing the AP while two STAs are moving away from it. Throughout the course of the simulation each STA remains within 35 m. of the AP so as to maintain the SNR required to support the data rate of 54 Mbps used in the simulation. The effects of extended ranges and their impact on the link SNR and data rate are explored in Chapter IV.

The traffic load on the network was configured specifically for this simulation using OPNET's standard application profiles. The specific types and durations of the network traffic emulated during the simulation are depicted in Figure 17. Note that *sta_1* conducts two video teleconferencing (VTC) sessions during the simulation. Each time, the STA randomly selects another STA in the WLAN to conduct the VTC session with since OPNET is not configured to use a server as the source or destination for VTC

session traffic. The profile of each network traffic type and its associated load is provided in Table 9. The OPNET standard FTP profile was altered somewhat to provide for larger file sizes, thus increasing the load on the network. The network traffic profiles outlined in Figure 17 were chosen as they represent the mix of high-rate data and multimedia traffic loads one might expect to see on an 802.11a WLAN.

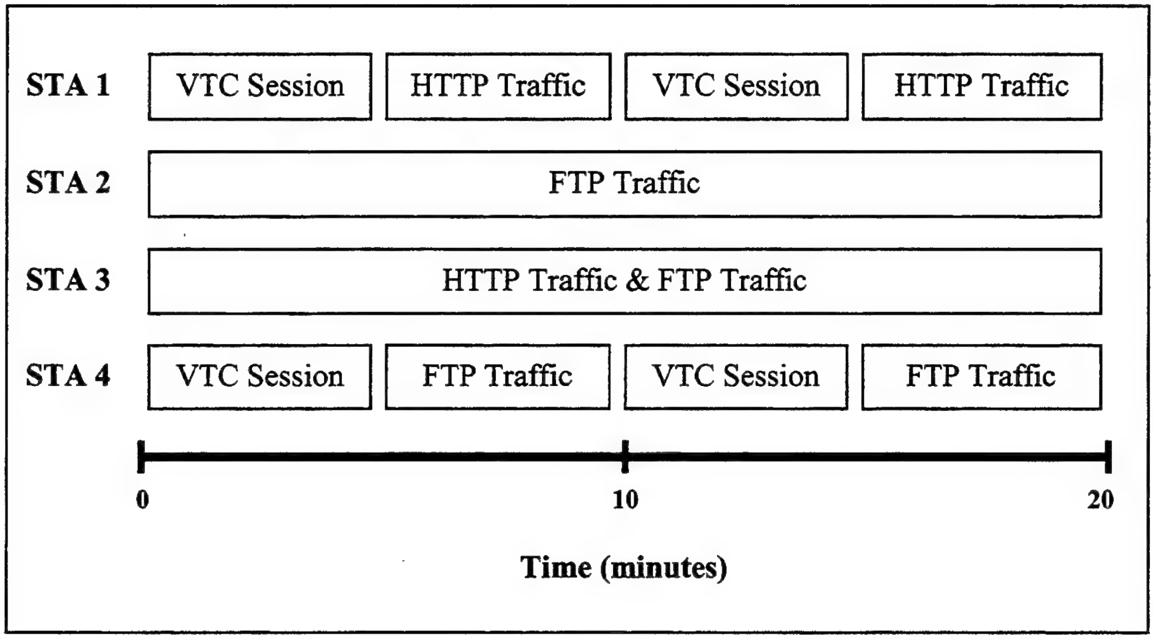

Figure 17. 802.11a Model Simulation Network Load Configuration.

Profile Name	Profile Attributes	Associated Data Rate (Mbps)
File Transfer (Heavy)	• Exponentially distributed random inter-request time (with a mean of 30 seconds) • Constant 125,000 byte file size	1 Mbps
Web Browsing (Heavy)	• Exponentially distributed random page interarrival time (with a mean of 20 seconds) • 1000 bytes of text per page • Five uniformly distributed images of size 500-2000 bytes per page	28 – 88 kbps
Video Conferencing (Light)	• Low resolution video • 10 frames per second • 128x120 pixels per frame • 9 bits per pixel	1.38 Mbps

Table 9. Simulated WLAN Traffic Profiles

The attributes and parameters of the STAs and AP were configured within the guidelines outlined earlier in this chapter. Specific WLAN settings used during the simulation are delineated in Table 10. These settings were applied to each STA and AP in the BSS. Channel 52 was chosen here since it is part of the middle UNII band and is ideal for use in a typical office environment. Note that the transmitter output power is set at the value specified for use in the middle UNII band (see Table 5). The RTS threshold was set at 500 to emulate the conditions found in a typical WLAN when RTS/CTS is enabled and the frame fragmentation option was disabled [26]. Finally, the data rate was set at the highest possible 802.11a value to test the model's operation at the fastest data rate. The simulation was conducted using OPNET version 7.0B on a Windows NT platform with a 366 MHz processor and 128 MB RAM.

WLAN Parameter	Setting
Data Rate	54 Mbps
Modulation Scheme	Ofdm_54Mbps
RTS Threshold (bytes)	500
Fragmentation Threshold (bytes)	None (disabled)
Bandwidth (kHz)	16,560
Base Frequency (MHz)	5251.7
Transmitter Output Power (W)	0.2
Receiver Noise Figure	5.01

Table 10. WLAN Attributes and Simulation Characteristics

The 802.11a baseline model simulation was completed successfully with a simulation duration of two hours and 11 minutes. A number of model performance statistics were collected by the OPNET simulation kernel during the trial. Of those, several are critical indicators used to determine that the model operated correctly. The total load on the WLAN as a function of time as the simulation progressed is one of the more important results. The overall WLAN load data is displayed in Figure 18, with the load given in bits per second. The results are as expected given the traffic profiles outlined in Table 9. The load on each STA and the AP is illustrated in Figure 19, where the effects of the differing traffic profiles are obvious. Also important in determining the successful operation of the MAC layer, its timing operations, and the RTS/CTS mechanism are the medium access delay and overall packet transmission delay statistics. Those results are displayed in Figure 20. The delay values increase with the load as we would expect, but do not exceed approximately 6 ms of overall delay and 3 ms of medium access delay. These values are typical of an operational WLAN under normal traffic loads [27].

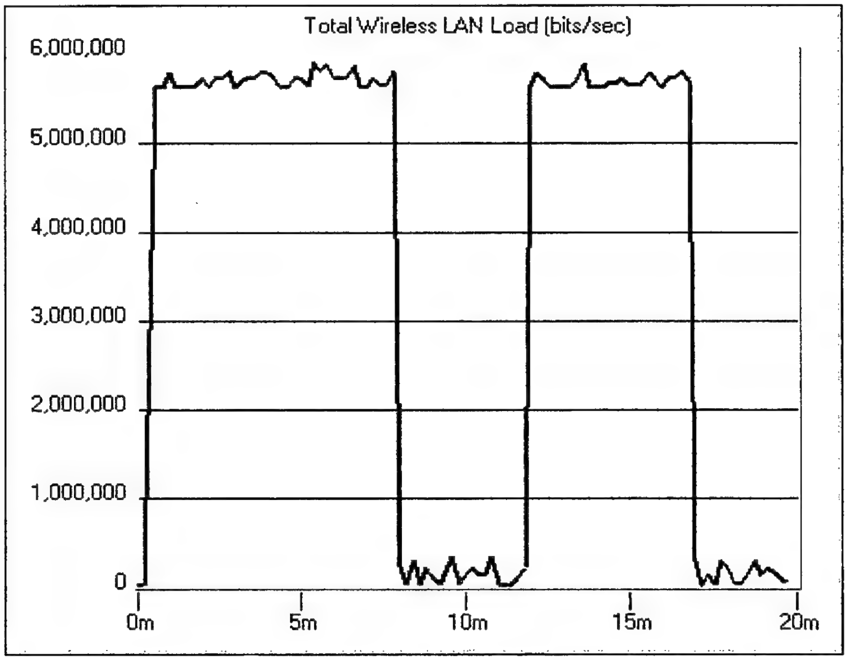

Figure 18. Total Load on the Simulated WLAN.

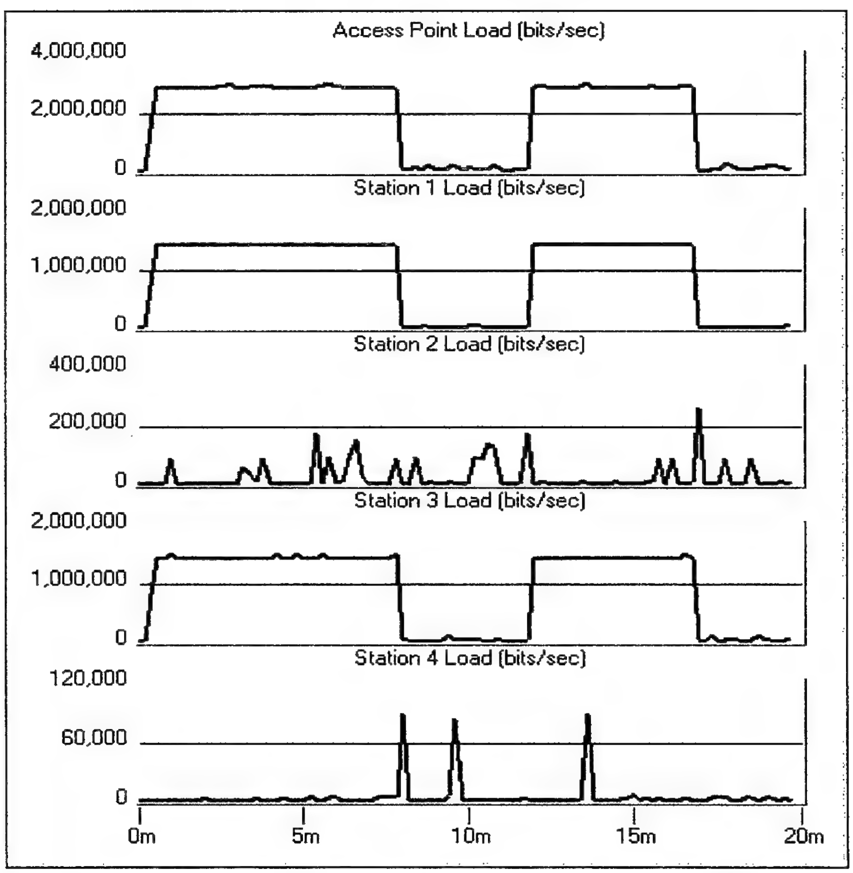

Figure 19. Individual Load Values for the AP and STAs.

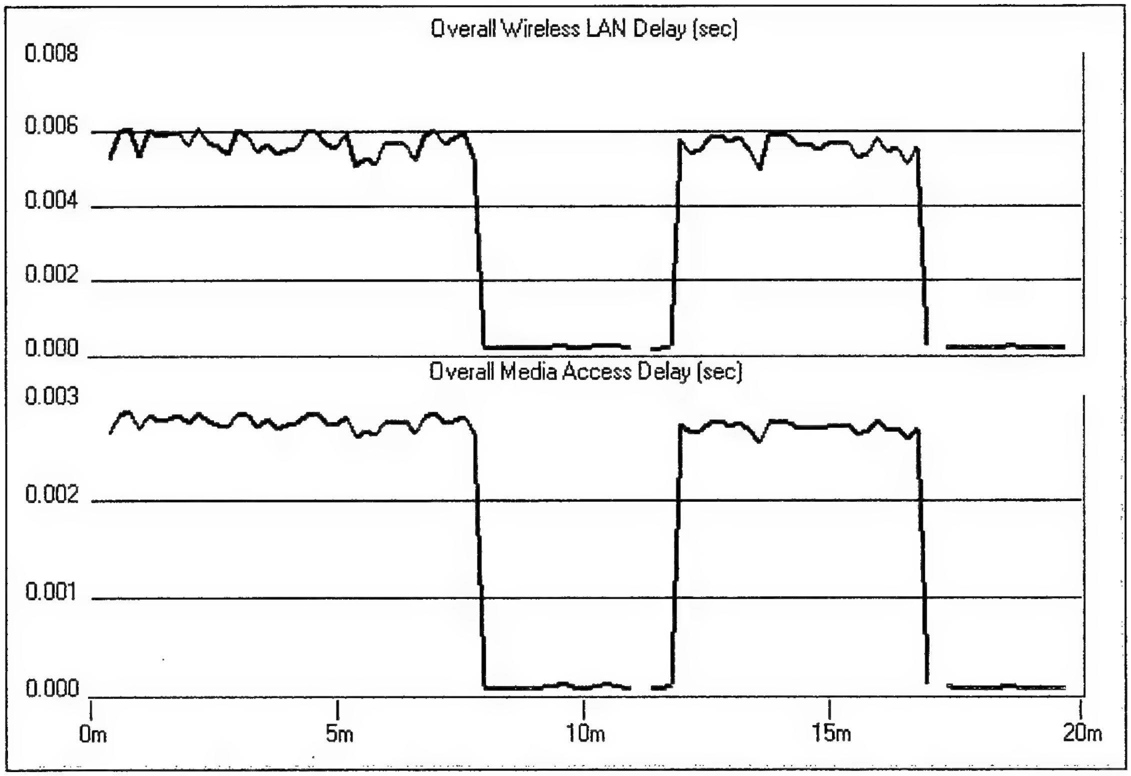

Figure 20. Simulated Medium Access Delay and Packet Delay.

The final results of the simulation are detailed in Figure 21 and they illustrate the link SNRs between the STAs and the AP. The values obtained here are roughly within the SNR ranges that might be expected in a typical hardware implementation given the range of SNRs seen in the OFDM modulation curves (Figure 14) and not accounting for hardware-specific gains and losses. In Figure 21 it is apparent that the SNR values change as expected when the STAs move closer to or farther away from the AP, thus reflecting the relationship between the inter-station distance and the received SNR. This signals that the internal mechanics of the model are indeed functioning correctly. In Chapter IV these SNR values will be utilized in one of the data rate agility mechanisms.

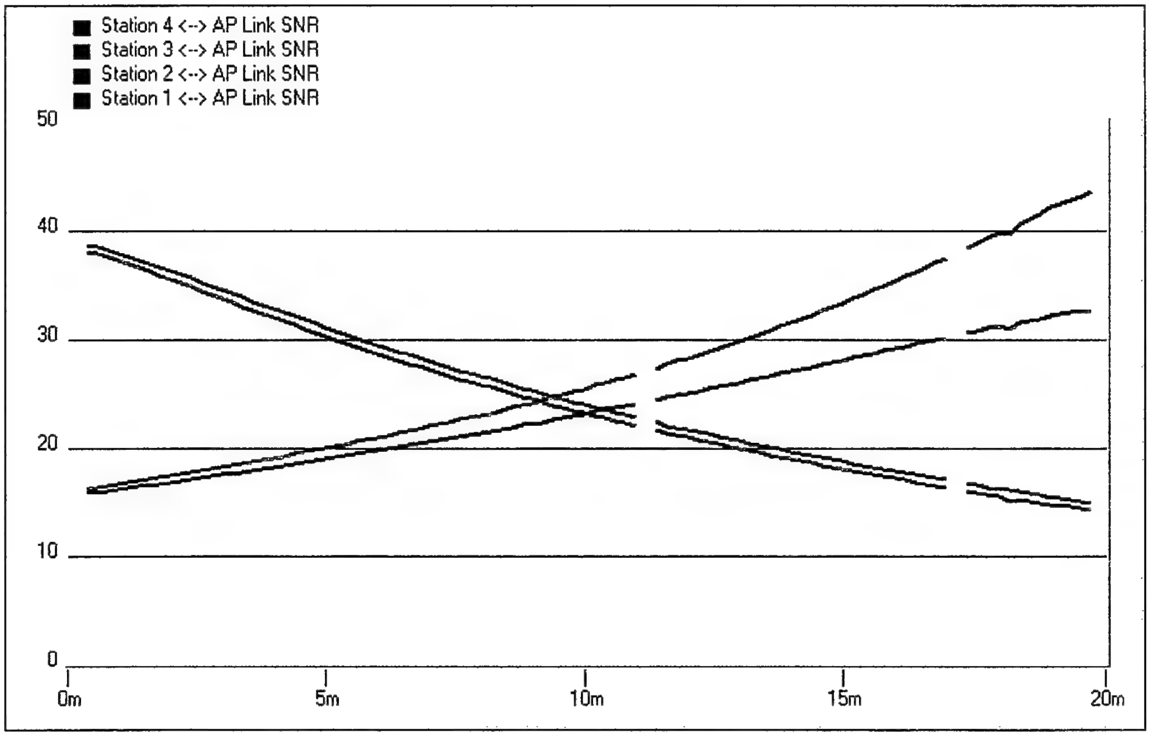

Figure 21. Simulated Link SNRs.

The results presented above indicate that the baseline 802.11a model introduced in section B of this chapter does function as it was designed to. The load, medium access delay, and SNR statistical results yielded by the simulation correspond to values that are characteristic of those obtained using fielded WLAN systems. This model is the first (that the author is aware of) to successfully wed a robust PHY layer implementation of the 802.11a protocol with the 802.11a MAC using the OPNET simulation tool to create a comprehensive 802.11a protocol model.

A model of an 802.11a-compliant WLAN constructed using the OPNET modeling and simulation tool was presented in this chapter. The composition of the MAC and PHY layers of the model were outlined in detail. The simulation results obtained using the model in standard network traffic conditions were presented as a measure of the model's validity. Now that the characteristics of the baseline model have been outlined and the model successfully tested, it will be used to explore several mechanisms through which an 802.11a WLAN implementation might dynamically alter its data rate based on the wireless link conditions.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DATA RATE AGILITY AND THE 802.11A BASELINE MODEL

The 802.11a standard does not specify a mechanism through which a WLAN implementation should dynamically alter the transmission data rate in response to changing link conditions. The standard does explicitly state that such a mechanism is allowed for; however, the exact mechanics are beyond the scope of the protocol. Most of the fielded 802.11 and 802.11b WLAN systems available today advertise data rate agility, but the specific mechanisms they use to do so are proprietary and unavailable for examination except at the most cursory level. The 802.11a standard and accompanying high data rates promise wireless access to multimedia applications with performance levels that parallel wired networks. Dynamic data rate agility and when or how it is implemented are therefore extremely important to the end-user as altering data rates may restrict access to the data associated with certain high-traffic applications. Two mechanisms for dynamically altering the data rate of an 802.11a WLAN are presented in the next section using the baseline 802.11a OPNET model outlined in Chapter III. The simulation results obtained using each mechanism are then presented and compared in Section B.

A. DATA RATE AGILITY MECHANISMS

802.11 and 802.11b WLAN implementations available on the market today typically include some permutation of data rate agility to reduce the operational speed of the WLAN in deteriorating link conditions. Particular examples include the popular Lucent ORiNOCO system and the Aironet 4000-series WLAN components [25, 28]. The specific techniques used to alter data rates in a given WLAN are commonly realized through a combination of both hardware and software approaches that are implementation specific. For instance, in a general sense Lucent's ORiNOCO system uses the link SNR after decorrelation of the spread spectrum signal in conjunction with receiver antenna diversity [21]. Additional details concerning specific vendor implementations are proprietary and were unavailable to the author.

Wireless networks based on the 802.11a protocol are likely to implement similar rate agility mechanisms. Designers of 802.11a-based WLAN implementations are still experimenting with rate agility techniques and algorithms in an effort to develop optimal adaptation mechanisms [29]. Two dynamic data rate agility mechanisms are presented in this chapter and implemented using the 802.11a baseline model. The first mechanism is

based on the link SNR while the second is based on the frame loss rate at the transmitting STA. Each technique is first explained and then simulation results obtained using each are presented to compare the two. In reality these two mechanisms are not mutually exclusive and they would likely be used in tandem to present the best possible criteria for rate adaptation. However, this analysis will examine the performance of each in isolation to measure their potential contributions to an inclusive dynamic data rate agility mechanism.

1. Rate Agility Based on Link SNR

An important indicator used in determining the quality of a wireless data link is the SNR of the transmission as measured at the receiver. Lower measured SNR values correspond to a higher probability of bit errors in the received frame. This relationship between the SNR and BER is reflected in the series of OFDM modulation curves outlined in Chapter III and presented in Figure 14. Higher bit error rates adversely impact both the PHY and MAC layers' ability to detect and correct errors using the MPDU cyclic redundancy code (CRC) and the FEC capability provided by convolutionally coded OFDM. High BERs translate to MAC failures and thus a cessation of successful frame exchange. The BER of a given wireless link must remain below a certain threshold for two STAs to effectively communicate. This threshold is typically on the order of 1×10^{-5} and a number of 802.11 and 802.11b implementations available today guarantee BERs below that threshold during a successful data exchange [25].

The relationship between the SNR and BER can be exploited to determine the quality of a wireless link by measuring the SNR at the receiver and using that measurement to adapt the link data rate to the link's current environmental conditions. If the SNR decreases there will likely be a corresponding increase in the BER. A STA can then alter the modulation scheme and coding rate combination of subsequent transmissions to provide a more robust, albeit lower data rate, transmitted signal. The converse is also true: higher received SNR values indicate improving link conditions. A STA can similarly use that information to alter its modulation and coding combination to realize a less robust but higher data rate link.

There are a number of choices that must be made concerning the use of received SNR values to adapt the wireless link data rate to changing link conditions. Either the most current SNR value could be used to alter the data rate, or the STA could use the trend in SNR values over a specified time period to adapt its data rate. The capabilities of

the STA transceiver hardware certainly impact this choice, as does the throughput of the link at the specified time. A link with a throughput of hundreds of packets per second provides a larger data set from which to make a rate agility decision while throughputs of only a packet or so per second provide far fewer decision points. The SNR-based mechanism presented and implemented here utilizes the instantaneous received SNR value to judge the link quality and alter the data rate accordingly (if necessary) rather than tracking the SNR over time.

If the received SNR is in fact either high or low enough to necessitate a change in the data rate, the STA must decide how much the rate should be increased or decreased based on the difference in SNR between the current frame and the previous frame. For example, suppose a frame is received with a much lower SNR value than that of the preceding frame. The STA might then either reduce its data rate to the next lower level or instantaneously reduce it to an even lower level as determined by a series of upper and lower SNR bounds associated with each data rate. In other words, the SNR-based data rate adaptation can be gradual or rapid. The former approach may be too slow in responding to the link conditions and might result in a link failure while the latter approach is susceptible to widely fluctuating SNR values where a STA could find itself continuously "chasing" the SNR. The mechanism presented in this thesis utilizes the latter technique. Given the low mobility of the STAs used in the simulation and the hierarchical distance vs. SNR relationship found in the OPNET pipeline stages, this approach was deemed to be the most effective for studying SNR-based data rate agility here.

There are obviously a number of alternative approaches to implementing an SNR-based dynamic data rate agility mechanism. To reiterate the techniques adopted for use in the mechanism analyzed here, the salient features of the SNR-based mechanism are summarized below:

- The SNR values utilized are associated with data and control frames only, not management traffic or other inter-STA communications.
- Rate agility is based on a single received SNR value and not SNR trends over time.
- The data rate can change to any other rate up to the maximum speed or down to the lowest mandatory speed without having to progress through any intermediary rates.

This SNR-based data rate agility mechanism was modeled by altering the OPNET 802.11a baseline model presented in Chapter III. A customized null data field called

"Link SNR" was added to both the *wlan_data_802_11a* and *wlan_control_802_11a* packet formats. This yielded two new packet formats: *wlan_data_802_11a_agility* and *wlan_control_802_11a_agility*. The "Link SNR" field is used by the *dra_snr_11a* pipeline stage (a modified version of *dra_snr*) to store the calculated SNR value for each individual frame that is transmitted through the wireless pipeline. When the frame is received at the destination, the SNR value is stripped from the "Link SNR" field at the MAC layer. The *wlan_mac_11a* process model was modified to include the data rate agility functionality that uses the SNR value once it is obtained from the pipeline stages. Specifically, the *wlan_physical_layer_data_arrival* function resident within *wlan_mac_11a* contains the proto-C code used to implement the rate agility mechanism. The modifications to the *wlan_mac_11a* code are procided in Appendix B and the *dra_snr_11a* pipeline stage code can be found in Appendix D.

A logical Proto-C code structure was added to *wlan_physical_layer_data_arrival* that compares the received SNR value to an upper and lower bound associated with each data rate. The lower bounds represent the minimum acceptable SNR required to keep the BER below 1×10^{-5} at each data rate while the upper bound of each rate is simply the lower bound of the next highest data rate. These data rate specific thresholds are provided in Table 11. They were selected based on the modulation curve values presented in Figure 14. However, special limits were constructed for the 9, 12, and 18 Mbps data rates based on the modulation curves. Both the 6 and 12 Mbps rates use BPSK and the 9 and 18 Mbps rates use QPSK, albeit at different coding rates. Since the probability of a bit error is equal for both BPSK and QPSK given a specific E_b/N_o value, the SNR range associated with the lower bound of the 9 Mbps rate and the upper bound of the 18 Mbps rate was divided into three equal ranges to represent the boundaries of the 9, 12, and 18 Mbps data rates [30]. These three ranges were used to provide for a hierarchical transition from the lowest data rate to the highest and vice versa.

Data Rate (Mbps)	SNR Lower Bound (dB)	SNR Upper Bound (dB)
6	4.38	5.38
9	5.38	5.84
12	5.84	6.30
18	6.30	6.76
24	6.76	8.86
36	8.86	9.70
48	9.70	12.22
54	12.22	N/A

Table 11. SNR Thresholds for Rate Agility.

Based on the comparison of the received SNR value with the rate-specific thresholds *wlan_mac_11a* selects the new operational speed at which that STA will transmit during its next frame transmission. In this manner a dynamic rate agility feature based on the instantaneous link SNR was added to the baseline 802.11a model. After the mechanism was implemented a new statistic collection vehicle was also added to *wlan_mac_11a* to track the data rate of the STA over time as link conditions change. This statistic enables the user to observe the data rate performance of the model during simulations. A performance analysis of the SNR-based mechanism is conducted in Section B of this chapter after presentation of the second data rate agility mechanism.

2. Rate Agility Based on the Frame Loss Rate

The number of frames dropped (i.e., lost) during the transmission process by a STA also serves as an excellent indicator of link performance in a WLAN. Frames can be dropped by a transmitting STA for one of two reasons. Firstly, the queue for frames awaiting transmission that were passed down from the higher layer can overflow resulting in frame losses. Secondly, the retransmission limit for a specific frame can be exceeded which will force the STA to cease its attempts to retransmit the frame and discard it. The former state, a higher layer buffer overflow, is caused by either a massive flood of data from the higher layer or a poor queue design in terms of capacity. Neither condition

speaks to the wireless link quality therefore packet losses resulting from a higher layer buffer overflow will not play a role in the mechanism presented here.

Frame retransmissions, however, are due to either collisions with other frames on the medium or a failure to receive an expected ACK or CTS frame in response to a transmitted data or RTS frame. Under typical WLAN operating conditions collisions do indeed take place, especially when there is a great deal of demand for the medium. Even in such high-load circumstances frame retry limits are rarely reached and the discarding of frames due to excessive retransmission attempts is extremely uncommon [26]. An increase in the number of frames dropped when the retransmission limits are exceeded is therefore a good indicator that the STA is failing to receive expected ACK and CTS packets. This in turn points to a deteriorating link since either the originating STA is not successfully transmitting data and RTS frames or the destination STA is not successfully transmitting ACK and CTS frames. The frame loss rate at a transmitting STA due to excessive retransmission attempts can therefore be used as the decision criteria in a dynamic data rate agility mechanism.

Rate agility cannot solely be based on the total number of frames dropped due to excessive retransmissions since under standard operating conditions a WLAN will eventually exceed any loss threshold. A specific time window must be delineated during which the number of lost frames should be tracked. If the frame loss rate exceeds a specified threshold (or series of thresholds) during that time window, then the data rate can be reduced accordingly. Too small a window selection may result in the STA responding prematurely to a transient environmental effect while too large a window may result in too slow of a response to a quickly deteriorating link. Additionally, if the traffic level is very low on the link then small data sets can result in disproportionate effects. For example, if only one packet traverses the link in the specified time window then the loss of that packet will translate to a 100% dropped frame rate and the data rate will automatically be reduced, regardless of whether the frame was dropped due to a transient effect or not. One second was chosen as the time window length for frame loss rate assessment in the mechanism implemented here. This selection reflects a balance between the possible effects given a low-traffic link and a reasonably quick response time to a truly deteriorating link.

A rate agility mechanism based on the frame loss rate must also be able to increase the data rate of the WLAN if the frame loss rate drops to an acceptable level. However, the realization of an acceptable frame loss rate does not necessarily imply that the link can sustain transmission at the next highest data rate. Perhaps acceptable frame

loss rates indicate that the WLAN is operating at the optimal data rate given the current link conditions. There are a number of possible approaches to the problem of increasing data rates after they have been reduced. These approaches include the addition of a second lower bound threshold below which the data rate would be increased, the use of a waiting period during which the STA cannot attempt to increase its data rates once it has reached a steady state, and simply allowing the STA to automatically attempt to raise the data rate immediately. Again, the goal is to prevent the STA from continuously oscillating back and forth between two adjacent data rates.

The second of the three approaches outlined above is implemented in the mechanism presented here. The author initially addressed the problem using the first approach, but the results obtained using a lower bound threshold were not promising. Those results are not provided here for the sake of brevity. Instead of using a lower threshold, the frame loss rate-based mechanism analyzed herein uses a steady-state waiting period during which the STA may not attempt to increase the data rate after it has been decreased. The drawback to this approach is that a STA might not be able to immediately take advantage of improving link conditions and increase the data rate; however, the advantage in terms of preventing rate oscillations outweighed the potential drawbacks.

The threshold for the acceptable frame loss rate was chosen based on the dropped frame rate observed in several trials conducted with the baseline 802.11a model. The first trial scenario consisted of a single high data rate link between a STA and an AP. The traffic profile was that of a continuous low-resolution VTC session between the mobile STA and a client terminal external to the BSS. The STA was given a trajectory that took it beyond the maximum allowable range for successful communications at 6 Mbps. A link failure condition was subsequently observed and the STA then moved back within range of the AP again. Figure 22 displays the number of packets dropped as the STA moved along its trajectory. The period of link failure during the session is clearly noticeable.

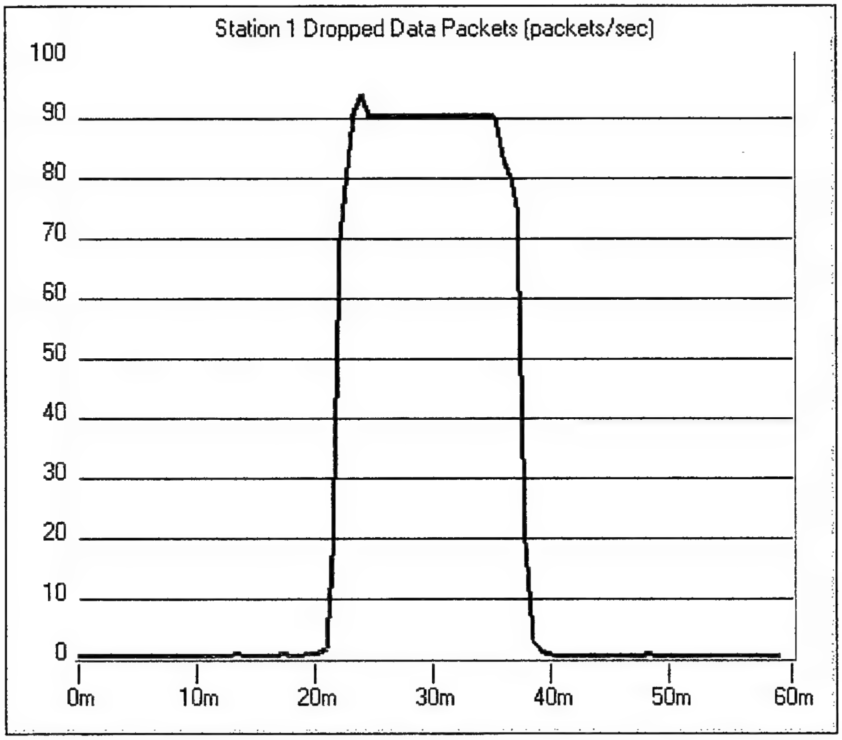

Figure 22. Simulated Packet Loss Rate in High Traffic Conditions.

The second scenario consisted of a lower data rate link between a STA and an AP. The traffic profile included heavy FTP and HTTP sessions between the STA and remote servers external to the BSS. The STA traversed the same trajectory utilized in the first scenario to include the link failure condition. A plot of the resulting number of packets dropped per second throughout the simulation is presented in Figure 23. The number of packets dropped during the link failure period in this trial is substantially lower than in the previous trial due to the lower traffic rate. These two trials were conducted to measure the packet loss rate under both high and low traffic conditions, as a data rate agility mechanism based on packet losses must function properly in both types of network load conditions.

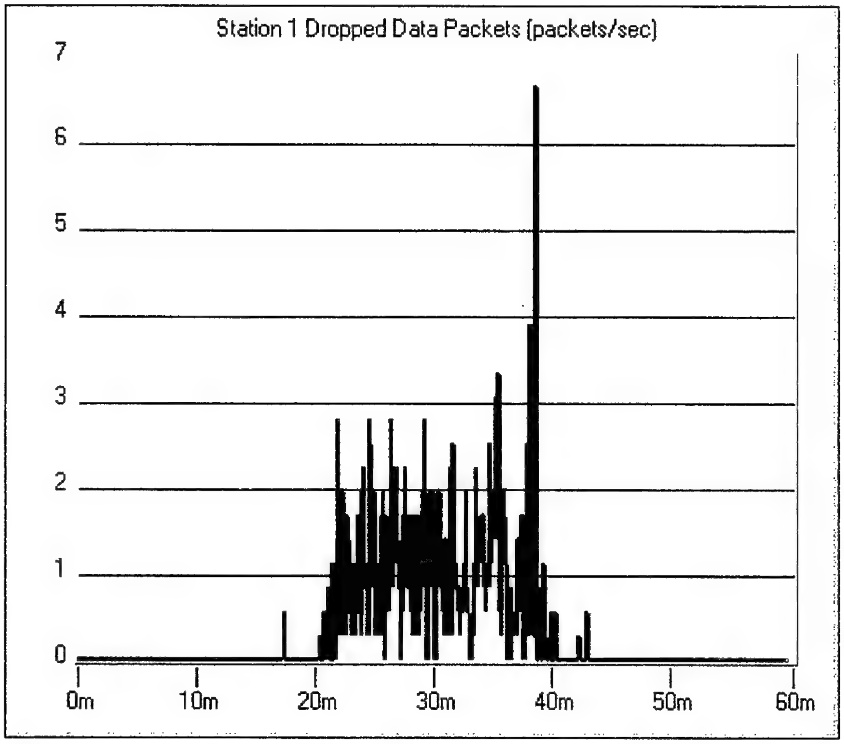

Figure 23. Simulated Packet Loss Rate in Low Traffic Conditions.

The data collected during the link failure condition in both trials was exported to a spreadsheet and the mean packet loss rate was calculated for each. The high-traffic link exhibited a mean packet loss rate of 88.487 packets per second and the low-traffic link was characterized by a mean packet loss rate of 1.073 packets per second. The standard deviation of the low-traffic link packet loss rate was found to be 0.6477. The lower loss rate is clearly the limiting factor in selecting proper thresholds for the rate agility mechanism as rate agility must be supported in both high and low traffic environments. Accordingly, a threshold value for the packet loss rate was selected as 0.437 packets per second, or one standard deviation below the mean packet loss rate associated with the low traffic simulation. When the frame loss rate increases above this threshold the transmission data rate is subsequently lowered.

Based on the results obtained during each trial and given the standard nature of the traffic load used to analyze the rate agility mechanisms, a frame loss rate of zero frames per second was selected as the frame loss rate that must be attained before a STA can seek to increase its data rate after the steady-state waiting period. Figures 22 and 23 both show the packet loss rate to be consistently zero under non-failure conditions. The steady-state waiting period was set at ten seconds in order to minimize data rate

oscillations. Whether the mechanism is increasing or decreasing the link data rate, the rate can only be increased or decreased one level at a time. This "stair step" approach is a by-product of only utilizing a single threshold to decrease the data rate.

There are obviously a number of alternative approaches to implementing a dynamic data rate agility mechanism based on the number of frames dropped per unit time. To reiterate the techniques adopted for use in the mechanism presented herein, the salient features of the rate agility mechanism based on the packet loss rate are summarized below:

- The time window used in determining the dropped frame rate is set at one second.
- The maximum frame loss rate threshold is set at 0.437 packets per second.
- The data rate can only "stair step" up one or down one level and cannot jump to and from non-adjacent rates.
- The acceptable frame loss rate that must be attained before a STA may seek to increase its data rate is zero frames per second.
- The steady-state waiting period that a STA must wait after attaining a zero frame loss rate before it attempts to increase its data rate is ten seconds.

This dropped frame-based rate agility mechanism was implemented by modifying the *wlan_mac_11a* process model to allow for rate agility based on the packet loss rate. Specifically, a counter was added to the *wlan_frame_discard function* to track the number of frames discarded and the threshold criteria, time window calculation, and steady-state waiting period timer were added to the body of the *wlan_prepare_frame_to_send* function. These changes were all made using Proto-C code logical structures and they are provided in Appendix C. Every time a frame is discarded by the MAC, the packet loss counter is incremented by one. During the subsequent frame transmission attempt, the packet loss counter is used in conjunction with the time window to calculate the current packet loss rate. The packet loss rate is then used in a comparison with the maximum packet loss rate threshold and the data rate of the STA is decreased accordingly (if necessary). The window size is then verified to be one second or less in size. If the time window exceeds one second, the packet loss counter is re-initialized so the packet loss rate can be refreshed for the subsequent interval.

If the packet loss rate reaches zero packets per second, the steady-state waiting period timer in *wlan_prepare_frame_to_send* is started. If the timer reaches ten seconds and the packet loss rate has remained at zero, the data rate is increased to the next highest

data rate up to the maximum rate as defined by the user. If the packet loss rate becomes non-zero before the timer has reached ten seconds, the timer is reset and the data rate may be decreased if the loss rate has exceeded the threshold. The same statistic collection vehicle used to track the data rate in the SNR-based mechanism was also added to this mechanism to provide the capability to monitor the data rate throughout the simulation. To obviate the effects of the possible effects of a higher layer queue overflow the buffer size was set to an artificially large value. Both the frame loss mechanism and the link SNR mechanism were employed in OPNET simulations. The simulation set-ups and the results are detailed in the next section.

B. RATE AGILITY MECHANISM SIMULATION RESULTS

The two dynamic data rate agility mechanisms detailed in section A were simulated in an 802.11a WLAN using OPNET. The simulations were conducted using an infrastructure BSS with a fixed AP and a single mobile STA. The STA was provided with a mobility profile that took it from a position adjacent to the AP along a straight path to a distance great enough to cause a link failure condition. The STA then reversed direction and returned to its original location. Each leg of the trajectory was 42 m long and the STA took 90 minutes to traverse it in each direction. The path that the STA traversed is depicted in Figure 24. These path lengths and mobility rates were chosen both to allow for a complete examination of the performance of each mechanism across the full spectrum of ranges expected in an 802.11a WLAN and to allow enough time for the WLAN to reach a theoretical steady state at each data rate during the course of the simulation.

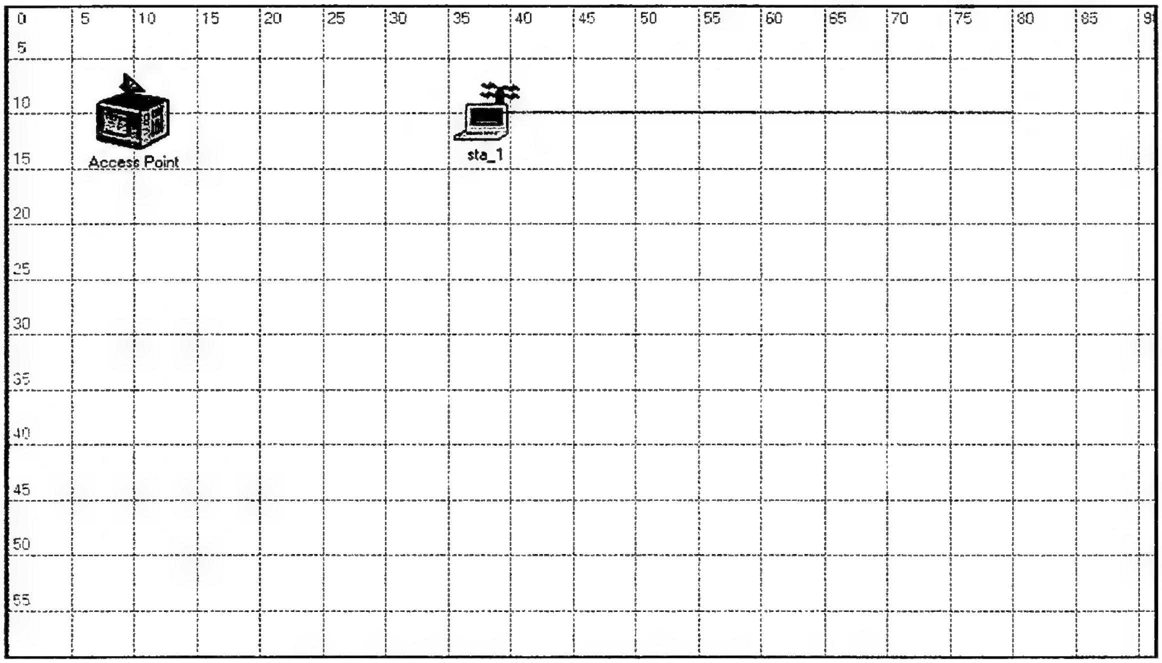

Figure 24. Rate Agility Simulation Environment.

Each rate agility mechanism was implemented in identical simulation environments using a traffic profile consisting of simultaneous heavy FTP and HTTP sessions with session parameters identical to those outlined in Chapter III, section C. This traffic profile was chosen because the rate agility mechanisms would be stressed to a greater extent given the fewer available data points characteristic of lower traffic loads. In other words, data rate agility is more difficult to implement in lower traffic environments than it is in high traffic conditions.

The simulation was first conducted with the SNR-based mechanism. With the traffic parameters and mobility profile outlined above the simulation duration was four hours and twenty-three minutes on the same machine used for the simulations detailed in Chapter III. The resulting data rate of the WLAN as a function of time is presented in Figure 25. These results closely follow the expected outcome, in that the data rate clearly drops level by level as the STA moves farther away from the AP. The period of link failure is clearly visible when the STA moves beyond the maximum range of the AP. The data rate subsequently increases again as the STA moves back toward the AP, eventually regaining the maximum 54 Mbps data rate. The BER and SNR values for transmissions on the link are depicted in Figures 26 and 27, respectively. The link SNR clearly drops as the STA moves away from the AP and then rises again as the STA closes on the AP while the BER remains at a minimum except during the link failure condition.

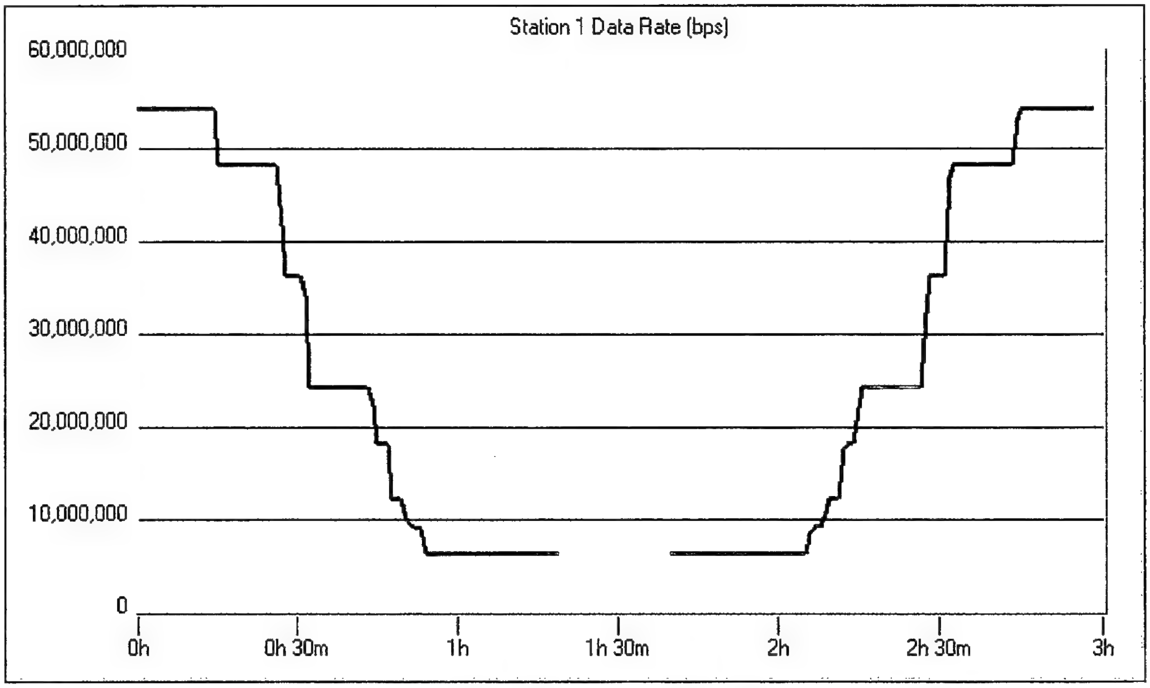

Figure 25. Simulated Data Rates with SNR-Based Rate Agility.

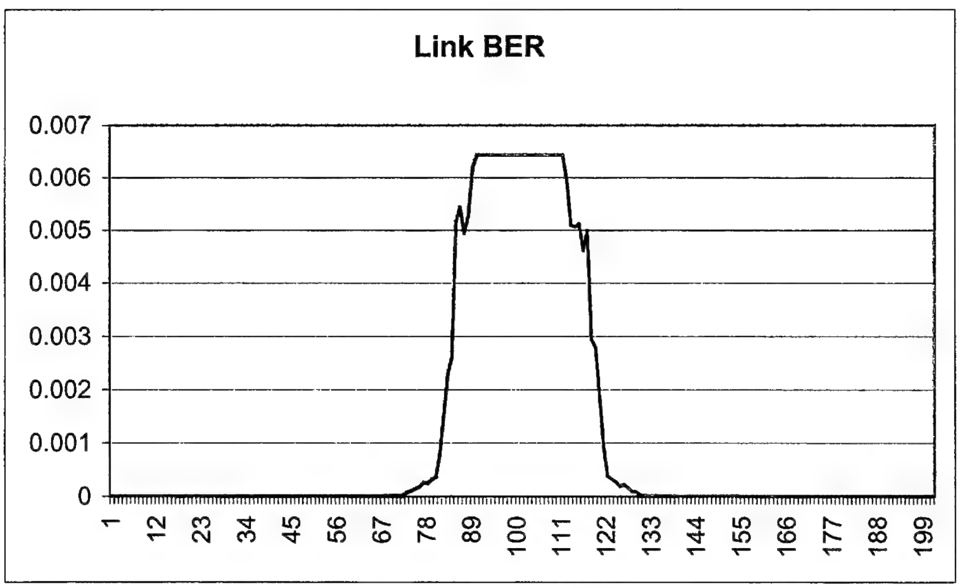

Figure 26. Simulated Link BERs with SNR-Based Rate Agility.

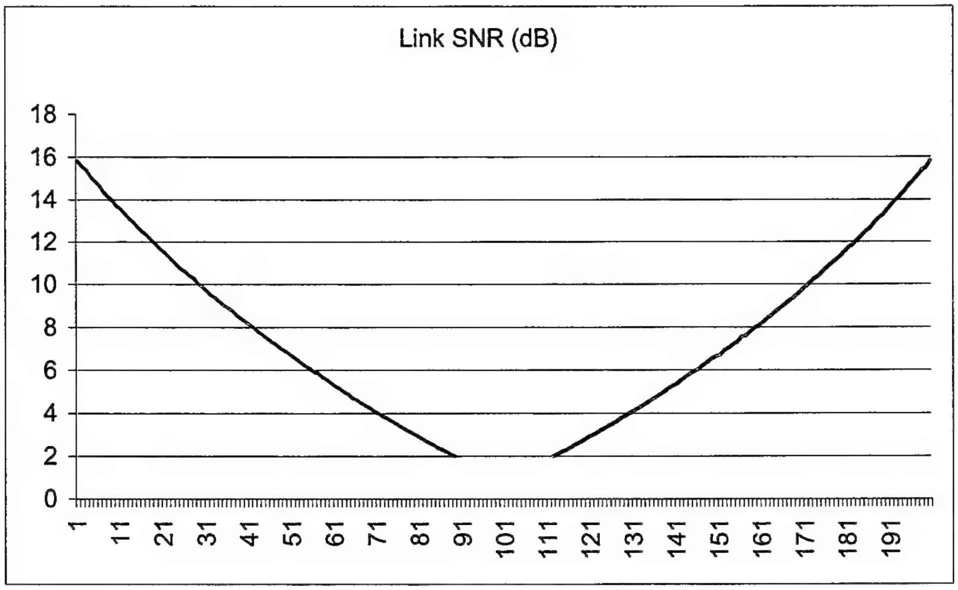

Figure 27. Simulated Link SNRs with SNR-Based Rate Agility.

These results suggest that the instantaneous SNR of a wireless networking link is an excellent criterion upon which to base dynamic data rate agility decisions in a WLAN implementation. Throughout the course of the simulation the WLAN MAC layer altered its data rates in a dynamic fashion to keep the link BER below 1×10^{-5} based on the received SNR value. The WLAN therefore avoided BERs that would necessitate a link failure while maintaining the highest possible data rate. The maximum operational range of the AP-STA link at each data rate are presented in Table 12. These range values were calculated using the data rate results from Figure 25 in conjunction with the mobility profile of the STA during the course of the simulation. Table 13 presents the nominal ranges of the 802.11b-compliant Lucent ORiNOCO PC card in a closed office environment for comparison. The simulation results support the claim of Atheros Communications, Inc. that 802.11a WLAN ranges will be comparable to those of 802.11b systems [14].

Data Rate (Mbps)	Range (m)
6	34.30
9	40.18
12	42.28
18	48.16
24	49.42
36	51.10
48	52.36
54	54.96

Table 12. 802.11a Ranges with SNR-Based Rate Agility.

Data Rate (Mbps)	Range (m)
1	50
2	40
5.5	35
11	25

Table 13. Nominal Ranges of the Lucent ORiNOCO PC Card (After Ref. [25]).

A simulation was then conducted using a WLAN implementation with data rate agility based on the frame loss rate mechanism. Again, the same traffic profile and STA trajectory were used in this simulation. The simulation duration in this instance was four hours and 48 minutes using the same machine. The data rates of the mobile STA obtained during the course of the simulation are presented in Figure 28. Although the data rate is 54 Mbps as expected at the start of the simulation, there is wide variation in the observed data rates for the remainder of the trial. The general trend in data rates matches those expected given the STA's trajectory, in that the data rate starts high, drops to 6 Mbps around the period of the link failure and then increases back to 54 Mbps at the conclusion of the simulation. The data rate results obtained using this mechanism are not stable enough to use in calculating the WLAN range per data rate.

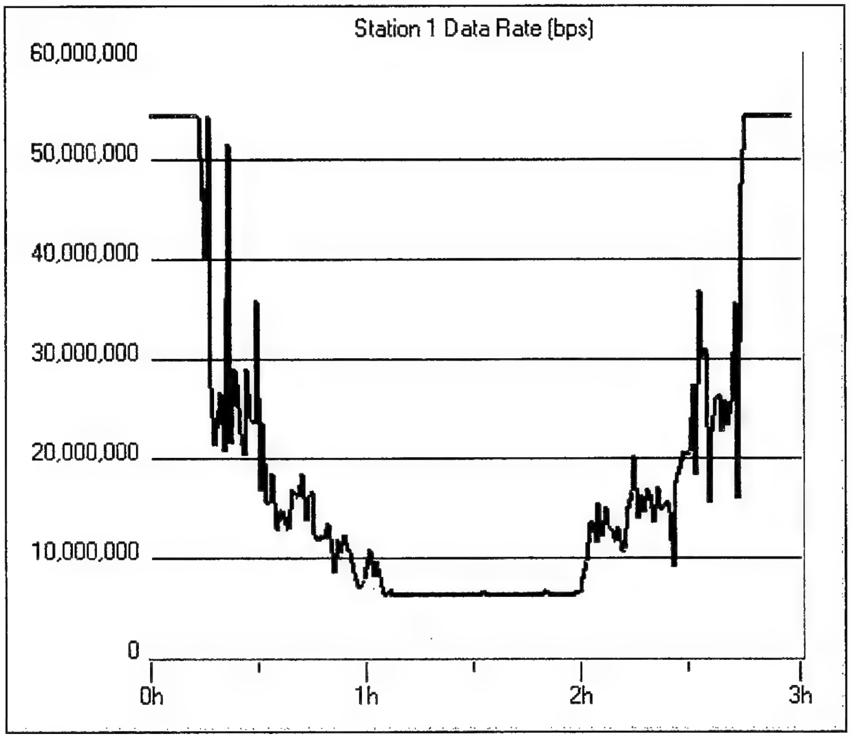

Figure 28. Simulated Data Rates with Frame Loss Rate-Based Agility.

In Figure 28 it is clear that the data rate drops off quickly once the STA exceeds the range of the 54 Mbps data rate and is slow to increase again as the STA moves closer to the AP. Based on these results it seems as if the rate agility mechanism based on the frame loss rate tends to underestimate the link quality and thus delivers lower data rates. This can be seen in Figure 29 where the two resultant data rate curves are shown together. The mean data rate for this trial using the frame loss rate mechanism (calculated with the numerical data used to construct Figure 28) was 20.135 Mbps. The mean data rate obtained during the simulation conducted with the SNR-based rate agility mechanism was 26.923 Mbps (calculated with the numerical data used to construct Figure 25). The SNR-based rate agility mechanism was able to produce a higher mean data rate over the course of the trial.

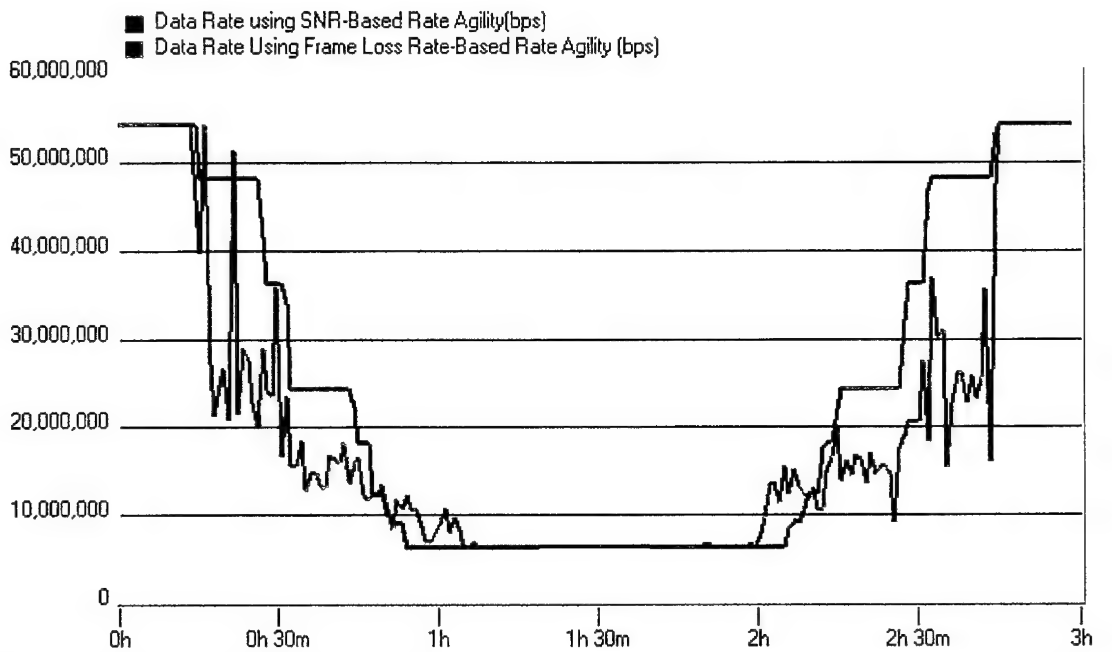

Figure 29. Data Rates for Both Mechanisms.

In this chapter two dynamic data rate agility mechanisms designed to allow for adaptive data rate behavior in a WLAN based on the link conditions were presented. The implementations of these mechanisms using the baseline OPNET 802.11a model were then outlined. The data rate results obtained using each mechanism were provided to allow for direct comparison of each method under the same simulated network traffic conditions. The results indicated that the mechanism based on the link SNR provides for the highest mean data rates and the smoothest data rate transitions. Conclusions and recommendations for further research are presented in Chapter V.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The results presented in this thesis indicate that the link SNR is a better criterion than the packet loss rate upon which to base dynamic data rate agility decisions in IEEE 802.11a-compliant WLANs. The simulation results obtained using the rate agility mechanism based on link SNR values illustrate smooth rate transitions during data rate increases and decreases as the link SNR changes in the presence of nominal WLAN traffic loads. The results obtained using the rate agility mechanism based on packet loss rates are characterized by a high degree of oscillation between data rates and a failure to reach steady-state data rates even during periods of unchanging link conditions. The SNR-based mechanism demonstrated a higher mean data rate over the course of the simulation. While packet loss rates do serve as a statistical indicator of adverse link conditions, the link SNR proved to be the superior criterion for use in 802.11a dynamic data rate agility mechanisms.

The 802.11a OPNET model used in the simulations conducted with each of the rate agility mechanisms includes both the MAC and PHY layers of the IEEE 802.11a standard. The MAC layer emulates each supported 802.11a data rate, correct medium access and transmission timing relationships, and the optional RTS/CTS mechanism. The PHY layer model includes the SNR versus BER characteristics of OFDM transmissions as reported in reference [17] and an experimentally determined path loss exponent found in reference [24]. The model's PHY layer does not include the Rayleigh-distributed fading losses typical of a wireless networking channel. Inclusion of these losses would perhaps effect the smooth data rate transitions seen with the SNR-based rate agility mechanism. The simulations also did not include the effects of random STA motion and varying mobility rates, nor did they account for the wide range of possible traffic profiles. These variables were not included for study due to the large simulation and computational overhead associated with higher traffic loads and longer simulation durations.

In reality, neither the link SNR nor the packet loss rate would be used in isolation to provide for dynamic data rate agility in a fielded 802.11a WLAN implementation. Both criteria would likely be combined with others, such as a comprehensive link history and hardware-specific attributes, to realize rate agility. Current 802.11b-compliant

WLANs utilize a variety of these techniques to achieve rate agility [21]. In any specific rate agility mechanism, trade-offs exist between responsiveness and rate vacillation and between providing the highest possible data rate and ensuring the robustness of the link. However, a direct comparison of SNR- and packet loss rate-based mechanisms for rate agility using the same model with identical traffic and mobility profiles indicated that link SNR is superior to the packet loss rate as a criterion for dynamic data rate agility in 802.11a WLANs.

B. RECOMMENDATIONS

The analysis presented in this thesis resulted from very specific data rate agility mechanisms implemented in a single model of the 802.11a protocol. The baseline 802.11a OPNET model itself could be further modified to provide a more detailed representation of the 802.11a protocol, or specific features of the model could be enhanced to further study a particular aspect of the protocol. The model presented herein includes a number of the MAC layer features developed at the Philips Research Labs; however, the PHY layer is a complete redesign of the PHY layer included in the OPNET 802.11 standard model. This comprehensive 802.11a model is the first to be developed (that the author is aware of) using the OPNET simulation environment. In addition, the PHY layer fidelity found in this model is rare given the infrequent application of OPNET to network protocol modeling at the PHY layer.

The 802.11a baseline model presented in Chapter III is a detailed model, but further modifications would only serve to increase its fidelity. The model could also be used in its current form to study other aspects of the 802.11a protocol and its behavior in specific network environments. Possibilities for additional research involving the baseline model include:

- Creating and including a variety of transceiver antenna designs and studying the effects of their transmission and reception patterns.
- The use of the OPNET Terrain Modeling Module (TMM) to explore the operational attributes of 802.11a WLANs in outdoor and tactical environments.
- The addition of a roaming and association feature and analysis of its performance.

- Analysis of WLAN performance as a function of the number of users in a BSS.
- Analysis of WLAN performance given a large number of users under varying traffic loads.
- Addition of the optional PCF medium access technique and an analysis of its performance in low latency traffic environment.
- Performance analysis of a WLAN given varying RTS and fragmentation thresholds.
- Addition of a Rayleigh fading channel loss model to the pipeline stages.

The model could also be applied in its current configuration to analyze different permutations of the two rate agility mechanisms presented in this thesis as well as to implement alternative rate agility techniques. Additional research opportunities for analyzing rate agility with this model include:

- Modification of the frame loss rate mechanism to base rate agility on the quantity of subsequent frame losses vice the loss rate over time.
- Modification of the SNR rate agility thresholds based on the addition of a Rayleigh fading channel model to the pipeline stages.
- Analysis of a rate agility mechanism based on the combination of the SNR and frame loss rate mechanisms.
- The use of transceiver antenna diversity in conjunction with a MAC-level mechanism to realize data rate agility.

The IEEE 802.11a WLAN protocol promises both mobility and the high data rate wireless connectivity required to deliver multimedia application traffic in a multi-user, multiple access environment. The 802.11a model presented in this thesis emulates the MAC and PHY layer behavior of the standard and provides the capability to conduct detailed investigations of the protocol's behavior. The model's applicability was demonstrated through the analysis of several dynamic data rate agility mechanisms in which the link SNR proved to be the most powerful indicator of link quality in the WLAN environment.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. WLAN_MAC_11A PROCESS MODEL OPNET CODE

This appendix provides the OPNET source code for the *wlan_mac_11a* process model. The code is an altered version of that found in the OPNET 802.11 standard model *wlan_mac* process model with additional features incorporated from the Philips Research Labs 802.11a OPNET model [1]. Comments applicable to the code modifications are included. The *wlan_mac_11a* STD was presented in Figure 11 and is reproduced below in Figure 30.

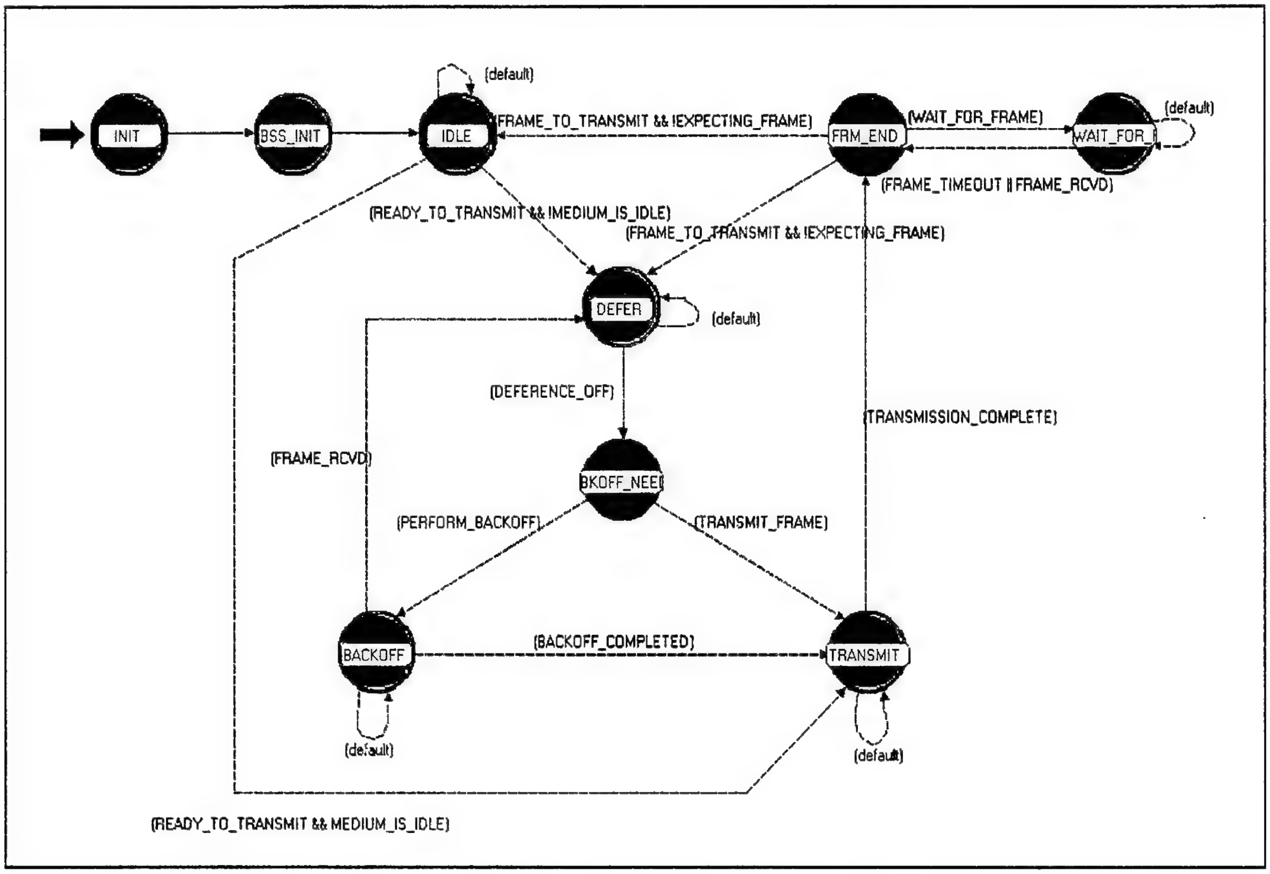

Figure 30. The *wlan_mac_11a* Process Model.

OPNET Code for *wlan_mac_11a*
Bryan E. Braswell
March 2001

Header Block

```
#include <math.h>
#include "oms_pr.h"
#include "oms_tan.h"
#include "oms_bgutil.h"

/* Definitions to all protocol specific parameters. */

/*  802.11a Model Addition. */
/* Include an altered support file to eliminate the control packet */
/* durations and add the new lowest mandatory data rate.            */
#include "wlan_support_11a.h"

/* Station address assignment definitions. */
#include "oms_auto_addr_support.h"
#include "oms_dist_support.h"

/* incoming statistics and stream wires */
#define TRANSMITTER_BUSY_INSTAT          8
#define LOW_LAYER_INPUT_STREAM_CH4       7

/*  802.11a Model Addition */
/* There are now 8 outgoing streams, one for each 802.11a data rate. */
#define LOW_LAYER_OUT_STREAM_CH1         0
#define LOW_LAYER_OUT_STREAM_CH2         1
#define LOW_LAYER_OUT_STREAM_CH3         2
#define LOW_LAYER_OUT_STREAM_CH4         3
#define LOW_LAYER_OUT_STREAM_CH5         4
#define LOW_LAYER_OUT_STREAM_CH6         5
#define LOW_LAYER_OUT_STREAM_CH7         6
#define LOW_LAYER_OUT_STREAM_CH8         7

/* Flags to load different variables based on attribute settings. */
#define        WLAN_AP                   1
#define        WLAN_STA                  0

/* Dimension count for global per-stream statistics. */
#define        WLANC_STRM_STAT_DIM_COUNT 32

/* Stream index for packets without stream information.        */
#define        WLANC_STRM_UNSET          -1

/* Define interrupt codes for generating handling interrupts      */
/* indicating changes in deference, frame timeout which infers    */
/* that the collision has occurred, random backoff and transmission */
/* completion by the physical layer (self interrupts).            */
typedef enum WlanT_Mac_Intrpt_Code
```

```
	{
	WlanC_Deference_Off,	/* Deference before frame transmission	*/
	WlanC_Frame_Timeout,	/* No frame rcvd in set duration (infer collision)	*/
	WlanC_Backoff_Elapsed,	/* Backoff done before frame transmission	*/
	WlanC_CW_Elapsed	/* Backoff done after successful frame transmission */
	} WlanT_Mac_Intrpt_Code;

/* Define codes for data and managment frames use in DCF	*/
/* The code defined is consistent with IEEE 802.11 format	*/
/* There are 6 bits used to define the code and in the following	*/
/* enumeration the first 6 bits are used in the type field of the frame.	*/
typedef enum WlanT_Mac_Frame_Type
	{
	WlanC_Rts  = 0x6C,	/* Rts code set into the Rts control frame	*/
	WlanC_Cts  = 0x70,	/* Cts code set into the Cts control frame	*/
	WlanC_Ack  = 0x74,	/* Ack code set into the Ack control frame	*/
	WlanC_Data = 0x80,	/* Data code set into the Data frame	*/
	WlanC_None = 0x00	/* None type	*/
	} WlanT_Mac_Frame_Type;

/* Defining codes for the physical layer characteristics type	*/

/*  802.11a Model Addition  */
/* There is only one physical layer possible with 802.11a: OFDM.  */
typedef enum WlanT_Phy_Char_Code
	{
	WlanC_OFDM
	} WlanT_Phy_Char_Code;

/* Define a structure to maintain data fragments received by each	*/
/* station for the purpose of reassembly (or defragmentation)	*/
typedef struct WlanT_Mac_Defragmentation_Buffer_Entry
	{
	int	tx_station_address;	/* Store the station address of transmitting station */
	double	time_rcvd;	/* Store time the last fragment for this frame was received */
	Sbhandle	reassembly_buffer_ptr;	/* Store data fragments for a particular packet	*/
	} WlanT_Mac_Defragmentation_Buffer_Entry;

/* Define a structure to maintain a copy of each unique data frame	*/
/* received by the station. This is done so that the station can	*/
/* discard any additional copies of the frame received by it.	*/
typedef struct WlanT_Mac_Duplicate_Buffer_Entry
	{
	int	tx_station_address;	/* store the station address of transmitting station	*/
	int	sequence_id;	/* rcvd packet sequence id	*/
	int	fragment_number;	/* rcvd packet fragment number	*/
	} WlanT_Mac_Duplicate_Buffer_Entry;

/* This structure contains all the flags used in this process model to determine	*/
/* various conditions as mentioned in the comments for each flag.	*/
typedef struct WlanT_Mac_Flags
	{
	Boolean	data_frame_to_send;	/* Flag to check when station needs to transmit.	*/
```

```
        Boolean   backoff_flag;          /* Backoff flag is set when either the collision is      */
                                         /* inferred or the channel switched from busy to idle   */
        Boolean   rts_sent;              /* Flag to indicate that wether the Rts for this         */
                                         /* particular data frame is sent                         */
        Boolean   rcvd_bad_packet;       /* Flag to indicate that the received packet is bad     */
        Boolean   receiver_busy;         /* Set this flag if receiver busy stat is enabled        */
        Boolean   transmitter_busy;      /* Set this flag if we are transmitting something.       */
        Boolean   wait_eifs_dur;         /* Set this flag if the station needs to wait for eifs   */
                                         /* duration.                                             */
        Boolean   gateway_flag;          /* Set this flag if the station is a gateway.            */
        Boolean   bridge_flag;           /* Set this flag if the station is a bridge              */
        Boolean   immediate_xmt;         /* Set this flag if the new frame can be transmitted     */
                                         /* without deferring.                                    */
        Boolean   cw_required;           /* Indicates the arrival of an ACK making the            */
                                         /* transmission successful. Requires a CW period.        */
        Boolean   nav_updated;           /* Indicates a new NAV value since the last time         */
                                         /* when self interrupt is scheduled for the end of       */
                                         /* deference.                                            */
        } WlanT_Mac_Flags;

/* This structure contains the destination address to which the received */
/* data packet needs to be sent and the contents of the recieved packet */
/* from the higher layer.
            */
typedef struct WlanT_Hld_List_Elem
        {
        double    time_rcvd;             /* Time packet is received by the higher layer    */
        int       destination_address;   /* Station to which this packet needs to be sent  */
        Packet*   pkptr;                 /* store packet contents                          */
        } WlanT_Hld_List_Elem;

/* Statistic handle array for dimensioned per-stream statistics.          */
typedef Stathandle        WlanT_Shandle_Array [WLANC_STRM_STAT_DIM_COUNT];

/* Boolean array that stores the registration status of per-stream statistics.       */
typedef Boolean           WlanT_Stat_Reg_Status_Array [WLANC_STRM_STAT_DIM_COUNT];

/**       Macros  Definition                                                  **/
/** The data frame send flag is set whenever there is a data to be send by    **/
/** the higher layer or the response frame needs to be sent. However,in **/
/** either case the flag will not be set if the receiver is busy             **/
/** Frames cannot be transmitted until medium is idle. Once, the medium      **/
/** is available then the station is eligible to transmit provided there     **/
/** is a need for backoff. Once the transmission is complete then the        **/
/** station will wait for the response provided the frame transmitted        **/
/** requires a response (such as Rts and Data frames). If response           **/
/** is not needed then the station will defer to transmit next packet        **/

/* After receiving a stream interrupt, we need to switch states from     */
/* idle to defer or transmit if there is a frame to transmit and the     */
/* receiver is not busy                                                  */
#define READY_TO_TRANSMIT            ((intrpt_type == OPC_INTRPT_STRM || (intrpt_type ==
                                     OPC_INTRPT_SELF && intrpt_code ==
```

```
                                        WlanC_CW_Elapsed)) && \
                                        (wlan_flags->data_frame_to_send ==
                                        OPC_BOOLINT_ENABLED || fresp_to_send !=
                                        WlanC_None) && \
                                        wlan_flags->receiver_busy == OPC_BOOLINT_DISABLED
                                        && \
                                        (current_time >= cw_end || fresp_to_send != WlanC_None))

/* When we have a frame to transmit, we move to transmit state if the    */
/* medium was idle for at least a DIFS time, otherwise we go to defer    */
/* state.
                                        */
#define MEDIUM_IS_IDLE                  ((current_time - nav_duration >= difs_time) && \
                                         (wlan_flags->receiver_busy ==
                                         OPC_BOOLINT_DISABLED) && \
                                        (current_time - rcv_idle_time >= difs_time))

/* Change state to Defer from Frm_End, if the input buffers are not empty or a frame needs      */
/* to be retransmitted or the station has to respond to some frame.
                        */
#define FRAME_TO_TRANSMIT               ((wlan_flags->data_frame_to_send ==
                                        OPC_BOOLINT_ENABLED && current_time >= cw_end) ||
                                        fresp_to_send != WlanC_None || retry_count != 0)

/* After defering for either collision avoidance or interframe gap    */
/* the channel will be available for transmission                                          */
#define DEFERENCE_OFF                   (intrpt_type == OPC_INTRPT_SELF && \
                                        intrpt_code == WlanC_Deference_Off && \
                                        wlan_flags->receiver_busy ==
                                        OPC_BOOLINT_DISABLED)

/* Isssue a transmission complete stat once the packet has successfully */
/* been transmitted from the source station                             */
#define TRANSMISSION_COMPLETE           (intrpt_type == OPC_INTRPT_STAT && \
                                        op_intrpt_stat () == TRANSMITTER_BUSY_INSTAT)

/* Backoff is performed based on the value of the backoff flag.              */
#define PERFORM_BACKOFF                 (wlan_flags->backoff_flag == OPC_BOOLINT_ENABLED)

/* Need to start transmitting frame once the backoff (self intrpt) completed          */
#define BACKOFF_COMPLETED               (intrpt_type == OPC_INTRPT_SELF && \
                                                intrpt_code == WlanC_Backoff_Elapsed && \
                                                wlan_flags->receiver_busy ==
                                                OPC_BOOLINT_DISABLED)

/* After transmission the station will wait for a frame response for       */
/* Data and Rts frames.                                                     */
#define WAIT_FOR_FRAME          (expected_frame_type != WlanC_None)

/* Need to retransmit frame if there is a frame timeout and the    */
/* required frame is not received                                  */
#define FRAME_TIMEOUT           (intrpt_type == OPC_INTRPT_SELF && intrpt_code ==
                                WlanC_Frame_Timeout)
```

```
/* If the frame is received appropriate response will be transmitted */
/* provided the medium is considered to be idle */
#define FRAME_RCVD                (intrpt_type == OPC_INTRPT_STRM && wlan_flags-
                                  >rcvd_bad_packet == OPC_BOOLINT_DISABLED && \
                                  i_strm <= LOW_LAYER_INPUT_STREAM_CH4)

/* Skip backoff if no backoff is needed */
#define TRANSMIT_FRAME            (wlan_flags->backoff_flag == OPC_BOOLINT_DISABLED)

/* Expecting frame response after data or Rts transmission */
#define EXPECTING_FRAME           (expected_frame_type != WlanC_None)

/* Macros that check the change in the busy status of the receiver. */
#define  RECEIVER_BUSY_HIGH       (intrpt_type == OPC_INTRPT_STAT &&
                                  intrpt_code < TRANSMITTER_BUSY_INSTAT && \
                                  op_stat_local_read (intrpt_code) == 1.0 &&
                                  (rcv_channel_status ^ (1 << intrpt_code) == 0))
#define  RECEIVER_BUSY_LOW        (intrpt_type == OPC_INTRPT_STAT && intrpt_code <
                                  TRANSMITTER_BUSY_INSTAT && \
                                  rcv_channel_status == 0)

/* Function declarations. */
static void               wlan_mac_sv_init ();
static void               wlan_higher_layer_data_arrival ();
static void               wlan_physical_layer_data_arrival ();
static void               wlan_hlpk_enqueue (Packet* hld_pkptr, int dest_addr);
Boolean                   wlan_tuple_find (int sta_addr, int seq_id, int frag_num);

static void               wlan_data_process (Packet* seg_pkptr, int sta_addr, int
final_dest_addr, int frag_num, int more_frag, int pkt_id, int rcvd_sta_bssid);

static void               wlan_accepted_frame_stats_update (Packet* seg_pkptr);
static void               wlan_per_stream_stat_register (int stream_index);
static void               wlan_interrupts_process ();
static void               wlan_prepare_frame_to_send (int frame_type);
static void               wlan_frame_transmit ();
static void               wlan_schedule_deference ();
static void               wlan_frame_discard ();
static void               wlan_mac_rcv_channel_status_update (int channel_id);
static void               wlan_mac_error (char* msg1, char* msg2, char* msg3);

/*  802.11a Model Addition  */
/* Add function for determining the control frame speed based on the operational data rate.  */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
static double             control_speed (double data_rate);

/*  802.11a Model Addition  */
/* Add function to compute the data field duration of an OFDM PPDU (in bits).  */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
static double          ppdu_duration (int PSDU_length, double transmission_rate);
```

State Variables Block

```
/* Internal state tracking for FSM */
FSM_SYS_STATE

/* State Variables */
int                                     retry_count;
int                                     intrpt_type;
WlanT_Mac_Intrpt_Code                   intrpt_code;
int                                     my_address;
List*                                   hld_list_ptr;
double                                  operational_speed;
int                                     frag_threshold;
int                                     packet_seq_number;
int                                     packet_frag_number;
int                                     destination_addr;
Sbhandle                                fragmentation_buffer_ptr;
WlanT_Mac_Frame_Type                    fresp_to_send;
double                                  nav_duration;
int                                     rts_threshold;
int                                     duplicate_entry;
WlanT_Mac_Frame_Type                    expected_frame_type;
int                                     remote_sta_addr;
double                                  backoff_slots;
Stathandle                              packet_load_handle;
double                                  intrpt_time;
Packet *                                wlan_transmit_frame_copy_ptr;
Stathandle                              backoff_slots_handle;
int                                     instrm_from_mac_if;
int                                     outstrm_to_mac_if;
int                                     num_fragments;
int                                     remainder_size;
List*                                   defragmentation_list_ptr;
WlanT_Mac_Flags*                        wlan_flags;
OmsT_Aa_Address_Handle                  oms_aa_handle;
double                                  current_time;
double                                  rcv_idle_time;
double                                  cw_end;
WlanT_Mac_Duplicate_Buffer_Entry**      duplicate_list_ptr;
Pmohandle                           hld_pmh;
int                                     max_backoff;
char                                    current_state_name [32];
Stathandle                              hl_packets_rcvd;
Stathandle                              media_access_delay;
Stathandle                              ete_delay_handle;
Stathandle                              global_ete_delay_handle;
Stathandle                              global_throughput_handle;
Stathandle                              global_load_handle;
Stathandle                              global_dropped_data_handle;
Stathandle                              global_mac_delay_handle;
Stathandle                              ctrl_traffic_rcvd_handle_inbits;
Stathandle                              ctrl_traffic_sent_handle_inbits;
Stathandle                              ctrl_traffic_rcvd_handle;
```

```
Stathandle                   ctrl_traffic_sent_handle;
Stathandle                   data_traffic_rcvd_handle_inbits;
Stathandle                   data_traffic_sent_handle_inbits;
Stathandle                   data_traffic_rcvd_handle;
Stathandle                   data_traffic_sent_handle;
WlanT_Shandle_Array          ete_delay_per_strm_sh_array;
WlanT_Shandle_Array          dropped_data_per_strm_sh_array;
WlanT_Shandle_Array          throughput_per_strm_sh_array;
WlanT_Stat_Reg_Status_Array  stat_reg_status_array;
double                       sifs_time;
double                       slot_time;
int                          cw_min;
int                          cw_max;
double                       difs_time;
Stathandle                   channel_reserv_handle;
Stathandle                   retrans_handle;
Stathandle                   throughput_handle;
int                          long_retry_limit;
int                          short_retry_limit;
int                          retry_limit;
WlanT_Mac_Frame_Type         last_frametx_type;
Evhandle                     deference_evh;
Evhandle                     backoff_elapsed_evh;
Evhandle                     frame_timeout_evh;
Evhandle                     cw_end_evh;
double                       eifs_time;
int                          i_strm;
Boolean                      wlan_trace_active;
int                          pkt_in_service;
Stathandle                   bits_load_handle;
int                          ap_flag;
int                          bss_flag;
int                          bss_id;
int                          hld_max_size;
double                       max_receive_lifetime;
WlanT_Phy_Char_Code          phy_char_flag;
OmsT_Aa_Address_Handle       oms_aa_wlan_handle;
int                          total_hlpk_size;
Stathandle                   drop_packet_handle;
Stathandle                   drop_packet_handle_inbits;
Log_Handle                   drop_pkt_log_handle;
Boolean                      drop_pkt_entry_log_flag;
int                          packet_size;
int                          packet_strm_id;
double                       receive_time;
Ici*                         llc_iciptr;
int                          rcv_channel_status;
int*                         bss_stn_list;
int                          bss_stn_count;
double                       plcp_preamble_duration;
double                       plcp_header_duration;
double                       plcp_overhead;
double                       response_speed;
```

```
} wlan_mac_11a_state;

#define pr_state_ptr                    ((wlan_mac_11a_state*) SimI_Mod_State_Ptr)
#define retry_count                     pr_state_ptr->retry_count
#define intrpt_type                     pr_state_ptr->intrpt_type
#define intrpt_code                     pr_state_ptr->intrpt_code
#define my_address                      pr_state_ptr->my_address
#define hld_list_ptr                    pr_state_ptr->hld_list_ptr
#define operational_speed               pr_state_ptr->operational_speed
#define frag_threshold                  pr_state_ptr->frag_threshold
#define packet_seq_number               pr_state_ptr->packet_seq_number
#define packet_frag_number              pr_state_ptr->packet_frag_number
#define destination_addr                pr_state_ptr->destination_addr
#define fragmentation_buffer_ptr        pr_state_ptr->fragmentation_buffer_ptr
#define fresp_to_send                   pr_state_ptr->fresp_to_send
#define nav_duration                    pr_state_ptr->nav_duration
#define rts_threshold                   pr_state_ptr->rts_threshold
#define duplicate_entry                 pr_state_ptr->duplicate_entry
#define expected_frame_type             pr_state_ptr->expected_frame_type
#define remote_sta_addr                 pr_state_ptr->remote_sta_addr
#define backoff_slots                   pr_state_ptr->backoff_slots
#define packet_load_handle              pr_state_ptr->packet_load_handle
#define intrpt_time                     pr_state_ptr->intrpt_time
#define wlan_transmit_frame_copy_ptr    pr_state_ptr->wlan_transmit_frame_copy_ptr
#define backoff_slots_handle            pr_state_ptr->backoff_slots_handle
#define instrm_from_mac_if              pr_state_ptr->instrm_from_mac_if
#define outstrm_to_mac_if               pr_state_ptr->outstrm_to_mac_if
#define num_fragments                   pr_state_ptr->num_fragments
#define remainder_size                  pr_state_ptr->remainder_size
#define defragmentation_list_ptr        pr_state_ptr->defragmentation_list_ptr
#define wlan_flags                      pr_state_ptr->wlan_flags
#define oms_aa_handle                   pr_state_ptr->oms_aa_handle
#define current_time                    pr_state_ptr->current_time
#define rcv_idle_time                   pr_state_ptr->rcv_idle_time
#define cw_end                          pr_state_ptr->cw_end
#define duplicate_list_ptr              pr_state_ptr->duplicate_list_ptr
#define hld_pmh                         pr_state_ptr->hld_pmh
#define max_backoff                     pr_state_ptr->max_backoff
#define current_state_name              pr_state_ptr->current_state_name
#define hl_packets_rcvd                 pr_state_ptr->hl_packets_rcvd
#define media_access_delay              pr_state_ptr->media_access_delay
#define ete_delay_handle                pr_state_ptr->ete_delay_handle
#define global_ete_delay_handle         pr_state_ptr->global_ete_delay_handle
#define global_throughput_handle        pr_state_ptr->global_throughput_handle
#define global_load_handle              pr_state_ptr->global_load_handle
#define global_dropped_data_handle      pr_state_ptr->global_dropped_data_handle
#define global_mac_delay_handle         pr_state_ptr->global_mac_delay_handle
#define ctrl_traffic_rcvd_handle_inbits pr_state_ptr->ctrl_traffic_rcvd_handle_inbits
#define ctrl_traffic_sent_handle_inbits pr_state_ptr->ctrl_traffic_sent_handle_inbits
#define ctrl_traffic_rcvd_handle        pr_state_ptr->ctrl_traffic_rcvd_handle
#define ctrl_traffic_sent_handle        pr_state_ptr->ctrl_traffic_sent_handle
#define data_traffic_rcvd_handle_inbits pr_state_ptr->data_traffic_rcvd_handle_inbits
#define data_traffic_sent_handle_inbits pr_state_ptr->data_traffic_sent_handle_inbits
```

```
#define data_traffic_rcvd_handle            pr_state_ptr->data_traffic_rcvd_handle
#define data_traffic_sent_handle            pr_state_ptr->data_traffic_sent_handle
#define ete_delay_per_strm_sh_array         pr_state_ptr->ete_delay_per_strm_sh_array
#define dropped_data_per_strm_sh_array      pr_state_ptr->dropped_data_per_strm_sh_array
#define throughput_per_strm_sh_array        pr_state_ptr->throughput_per_strm_sh_array
#define stat_reg_status_array               pr_state_ptr->stat_reg_status_array
#define sifs_time                   pr_state_ptr->sifs_time
#define slot_time                   pr_state_ptr->slot_time
#define cw_min                              pr_state_ptr->cw_min
#define cw_max                              pr_state_ptr->cw_max
#define difs_time                   pr_state_ptr->difs_time
#define channel_reserv_handle               pr_state_ptr->channel_reserv_handle
#define retrans_handle                      pr_state_ptr->retrans_handle
#define throughput_handle                   pr_state_ptr->throughput_handle
#define long_retry_limit                    pr_state_ptr->long_retry_limit
#define short_retry_limit                   pr_state_ptr->short_retry_limit
#define retry_limit                 pr_state_ptr->retry_limit
#define last_frametx_type                   pr_state_ptr->last_frametx_type
#define deference_evh                       pr_state_ptr->deference_evh
#define backoff_elapsed_evh                 pr_state_ptr->backoff_elapsed_evh
#define frame_timeout_evh                   pr_state_ptr->frame_timeout_evh
#define cw_end_evh                          pr_state_ptr->cw_end_evh
#define eifs_time                   pr_state_ptr->eifs_time
#define i_strm                              pr_state_ptr->i_strm
#define wlan_trace_active                   pr_state_ptr->wlan_trace_active
#define pkt_in_service                      pr_state_ptr->pkt_in_service
#define bits_load_handle                    pr_state_ptr->bits_load_handle
#define ap_flag                     pr_state_ptr->ap_flag
#define bss_flag                    pr_state_ptr->bss_flag
#define bss_id                              pr_state_ptr->bss_id
#define hld_max_size                        pr_state_ptr->hld_max_size
#define max_receive_lifetime                pr_state_ptr->max_receive_lifetime
#define phy_char_flag                       pr_state_ptr->phy_char_flag
#define oms_aa_wlan_handle                  pr_state_ptr->oms_aa_wlan_handle
#define total_hlpk_size                     pr_state_ptr->total_hlpk_size
#define drop_packet_handle                  pr_state_ptr->drop_packet_handle
#define drop_packet_handle_inbits           pr_state_ptr->drop_packet_handle_inbits
#define drop_pkt_log_handle                 pr_state_ptr->drop_pkt_log_handle
#define drop_pkt_entry_log_flag             pr_state_ptr->drop_pkt_entry_log_flag
#define packet_size                         pr_state_ptr->packet_size
#define packet_strm_id                      pr_state_ptr->packet_strm_id
#define receive_time                        pr_state_ptr->receive_time
#define llc_iciptr                          pr_state_ptr->llc_iciptr
#define rcv_channel_status                  pr_state_ptr->rcv_channel_status
#define bss_stn_list                        pr_state_ptr->bss_stn_list
#define bss_stn_count                       pr_state_ptr->bss_stn_count
#define plcp_preamble_duration              pr_state_ptr->plcp_preamble_duration
#define plcp_header_duration                pr_state_ptr->plcp_header_duration
#define plcp_overhead                       pr_state_ptr->plcp_overhead
#define response_speed                      pr_state_ptr->response_speed

/* This macro definition will define a local variable called      */
/* "op_sv_ptr" in each function containing a FIN statement.      */
```

```
/* This variable points to the state variable data structure,      */
/* and can be used from a C debugger to display their values.  */
#undef FIN_PREAMBLE
#define FIN_PREAMBLE wlan_mac_11a_state *op_sv_ptr = pr_state_ptr;
```

Temporary Variables Block

```
/* variables used for registering and discovering process models */
OmsT_Pr_Handle              process_record_handle;
List*                       proc_record_handle_list_ptr;
int                         record_handle_list_size;
int                         ap_count;
int                         count;
double                      sta_addr;
double                      statype ;
Objid                       mac_if_module_objid;
char                        proc_model_name [300];
Objid                       my_subnet_objid;
Objid                       my_objid;
Objid                       my_node_objid;
Objid                       params_attr_objid;
Objid                       wlan_params_comp_attr_objid;
Objid                       strm_objid;
int                         strm;
int                         i,j;
int                         addr_index;
int                         num_out_assoc;
int                         node_count;
int                         node_objid;
WlanT_Hld_List_Elem*        hld_ptr;
Prohandle                   own_prohandle;
double                      timer_duration;
double                      cw_slots;
char                        msg1 [120];
char                        msg2 [120];
WlanT_Phy_Char_Code         sta_phy_char_flag;
```

Function Block

```
static void
wlan_mac_sv_init ()
        {
        Objid                       mac_params_comp_attr_objid;
        Objid                       params_attr_objid;
        Objid                       phy_params_comp_attr_objid;
        Objid                       my_objid;
        Objid                       my_node_objid;
        Objid                       my_subnet_objid;
        Objid                       rx_objid;
        Objid                       tx_objid;
        Objid                       chann_params_comp_attr_objid;
```

```
Objid                                   subchann_params_attr_objid;
Objid                                   chann_objid;
Objid                                   sub_chann_objid;
int                                     num_chann;
char                                    subnet_name [512];
double                                  bandwidth;
double                                  frequency;
int                                     ap1_flag, i ;

/** 1. Initialize state variables.                                  **/
/** 2. Read model attribute values in variables.                    **/
/** 3. Create global lists                                          **/
/** 4. Register statistics handlers                                 **/
FIN (wlan_mac_sv_init ());

/* object id of the surrounding processor.      */
my_objid = op_id_self ();

/* Obtain the node's object identifier          */
my_node_objid = op_topo_parent (my_objid);

/* Obtain subnet objid.                                     */
my_subnet_objid = op_topo_parent (my_node_objid);

/* Obtain the values assigned to the various attributes */
op_ima_obj_attr_get (my_objid, "Wireless LAN Parameters", &mac_params_comp_attr_objid);
params_attr_objid = op_topo_child (mac_params_comp_attr_objid, OPC_OBJTYPE_GENERIC,
0);

/* Determine the assigned MAC address.      */
op_ima_obj_attr_get (my_objid, "station_address", &my_address);

/* Obtain an address handle for resolving WLAN MAC addresses.       */
oms_aa_handle = oms_aa_address_handle_get ("MAC Addresses", "station_address");

/* Creating a pool of station addresses for each subnet based on subnet name.       */
op_ima_obj_attr_get (my_subnet_objid, "name", &subnet_name);
oms_aa_wlan_handle = oms_aa_address_handle_get (subnet_name, "station_address");

/* Get model attributes.     */
op_ima_obj_attr_get (params_attr_objid, "Data Rate", &operational_speed);
op_ima_obj_attr_get (params_attr_objid, "Fragmentation Threshold", &frag_threshold);
op_ima_obj_attr_get (params_attr_objid, "Rts Threshold", &rts_threshold);
op_ima_obj_attr_get (params_attr_objid, "Short Retry Limit", &short_retry_limit);
op_ima_obj_attr_get (params_attr_objid, "Long Retry Limit", &long_retry_limit);
op_ima_obj_attr_get (params_attr_objid, "Access Point Functionality", &ap_flag);
op_ima_obj_attr_get (params_attr_objid, "Buffer Size", &hld_max_size);
op_ima_obj_attr_get (params_attr_objid, "Max Receive Lifetime", &max_receive_lifetime);

/* Initialize the retry limit for the current frame to long retry limit.        */
retry_limit = long_retry_limit;
```

```
/* Get the Channel Settings.
        */
/* Extracting Channel 0,1,2,3,4,5,6,7 (i.e. 6,9,12,18,24,36,48 and 54 Mbps) settings.      */
op_ima_obj_attr_get (params_attr_objid, "Channel Settings", &chann_params_comp_attr_objid);
subchann_params_attr_objid = op_topo_child (chann_params_comp_attr_objid,
OPC_OBJTYPE_GENERIC, 0);
op_ima_obj_attr_get (subchann_params_attr_objid, "Bandwidth", &bandwidth);
op_ima_obj_attr_get (subchann_params_attr_objid, "Min Frequency", &frequency);

/* Load the appropriate physical layer characteristics. Here, this will only be OFDM.  */
op_ima_obj_attr_get (params_attr_objid, "Physical Characteristics", &phy_char_flag);

/*  802.11a Model Addition  */
/* Based on physical charateristics of OFDM, set appropriate values for SIFS time,  */
/* Slot time, and the min/max contention window sizes.  */
/* Also, include values for the PLCP preamble and PLCP header (minus the SERVICE field)  */
/* in terms of seconds for use later.  */
switch (phy_char_flag)
        {
        case WlanC_OFDM:
                {
                /* Slot duration in units of sec.       */
                slot_time = .000009;

                /* Short interframe gap (SIFS) in units of sec.         */
                sifs_time = .000016;

                /* Minimum contention window size for selecting backoff slots.          */
                cw_min = 15;

                /* Maximum contention window size for selecting backoff slots.          */
                cw_max = 1023;

                /* PLCP Preamble in units of seconds.  */
                plcp_preamble_duration = .000016;

                /* PLCP Header (not including the SERVICE field) in units of seconds.  */
                plcp_header_duration = .000004;
                break;
                }

        default:
                {
                wlan_mac_error ("Unexpected Physical Layer Characteristic encountered.",
 OPC_NIL, OPC_NIL);
                break;
                }
        }

/*  802.11a Model Addition  */
/* Calculate the 802.11a PLCP overhead (preamble and header) in units of seconds.  */
plcp_overhead = plcp_preamble_duration + plcp_header_duration;
```

```
/* By default stations are configured for IBSS unless an Access Point is found,   */
/* then the network will have an infrastructure BSS configuration.                */
bss_flag = OPC_BOOLINT_DISABLED;

/* Computing DIFS interval which is interframe gap between successive      */
/* frame transmissions.                                                     */
difs_time = sifs_time + 2 * slot_time;

/*  802.11a Model Addition  */
/* If the receiver detects that the received frame is erroneous then it    */
/* will set the network allocation vector to EIFS duration.                */
/* The EIFS time for 802.11a is calculated per the 802.11 specification  */
/* (see Section 9.2.10, page 85) using the lowest mandatory data rate of 6 Mbps  */
eifs_time = difs_time + sifs_time + plcp_overhead + ppdu_duration (WLAN_ACK_LENGTH,
WLAN_MIN_MAN_DATA_RATE);

/* Creating list to store data arrived from higher layer.*/
hld_list_ptr = op_prg_list_create ();

/* Initialize segmentation and reassembly buffer.     */
defragmentation_list_ptr = op_prg_list_create ();
fragmentation_buffer_ptr = op_sar_buf_create (OPC_SAR_BUF_TYPE_SEGMENT,
OPC_SAR_BUF_OPT_PK_BNDRY);

/* Registering local statistics.      */
packet_load_handle              = op_stat_reg ("Wireless Lan.Load (packets)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
bits_load_handle                = op_stat_reg ("Wireless Lan.Load (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
hl_packets_rcvd                 = op_stat_reg ("Wireless Lan.Hld Queue Size (packets)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
backoff_slots_handle            = op_stat_reg ("Wireless Lan.Backoff Slots (slots)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
data_traffic_sent_handle        = op_stat_reg ("Wireless Lan.Data Traffic Sent (packets/sec)",
 OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
data_traffic_rcvd_handle        = op_stat_reg ("Wireless Lan.Data Traffic Rcvd
(packets/sec)",    OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
data_traffic_sent_handle_inbits = op_stat_reg ("Wireless Lan.Data Traffic Sent (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
data_traffic_rcvd_handle_inbits = op_stat_reg ("Wireless Lan.Data Traffic Rcvd (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
ctrl_traffic_sent_handle        = op_stat_reg ("Wireless Lan.Control Traffic Sent
(packets/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
ctrl_traffic_rcvd_handle        = op_stat_reg ("Wireless Lan.Control Traffic Rcvd
(packets/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
ctrl_traffic_sent_handle_inbits = op_stat_reg ("Wireless Lan.Control Traffic Sent (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
ctrl_traffic_rcvd_handle_inbits = op_stat_reg ("Wireless Lan.Control Traffic Rcvd (bits/sec)",
 OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
drop_packet_handle              = op_stat_reg ("Wireless Lan.Dropped Data Packets
(packets/sec)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
drop_packet_handle_inbits       = op_stat_reg ("Wireless Lan.Dropped Data Packets
(bits/sec)",        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
```

```
	retrans_handle			= op_stat_reg ("Wireless Lan.Retransmission Attempts
(packets)", OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	media_access_delay		= op_stat_reg ("Wireless Lan.Media Access Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	ete_delay_handle		= op_stat_reg ("Wireless Lan.Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	channel_reserv_handle		= op_stat_reg ("Wireless Lan.Channel Reservation (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
	throughput_handle		= op_stat_reg ("Wireless Lan.Throughput (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);

	/* Registering global statistics.	*/
	global_ete_delay_handle		= op_stat_reg ("Wireless LAN.Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
	global_load_handle		= op_stat_reg ("Wireless LAN.Load (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
	global_throughput_handle = op_stat_reg ("Wireless LAN.Throughput (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
	global_dropped_data_handle	= op_stat_reg ("Wireless LAN.Data Dropped (bits/sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
	global_mac_delay_handle		= op_stat_reg ("Wireless LAN.Media Access Delay (sec)",
OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

	/* Initialize the registration status array for per-stream*/
	/* statistics. We will register them only if we detect a */
	/* packet that belongs to that particular stream.			*/
	for (i = 0; i < WLANC_STRM_STAT_DIM_COUNT; i++)
		{
		stat_reg_status_array [i] = OPC_FALSE;
		}

	/* Registering log handles */
	drop_pkt_log_handle	= op_prg_log_handle_create (OpC_Log_Category_Configuration,
"Wireless Lan", "Data packet Drop", 128);
	drop_pkt_entry_log_flag = OPC_FALSE;

	/* Allocating memory for the flags used in this process model. */
	wlan_flags = (WlanT_Mac_Flags *) op_prg_mem_alloc (sizeof (WlanT_Mac_Flags));

	/* Disabling all flags as a default.	*/
	wlan_flags->data_frame_to_send		= OPC_BOOLINT_DISABLED;
	wlan_flags->backoff_flag		= OPC_BOOLINT_DISABLED;
	wlan_flags->rts_sent			= OPC_BOOLINT_DISABLED;
	wlan_flags->rcvd_bad_packet		= OPC_BOOLINT_DISABLED;
	wlan_flags->receiver_busy		= OPC_BOOLINT_DISABLED;
	wlan_flags->transmitter_busy		= OPC_BOOLINT_DISABLED;
	wlan_flags->gateway_flag		= OPC_BOOLINT_DISABLED;
	wlan_flags->bridge_flag			= OPC_BOOLINT_DISABLED;
	wlan_flags->wait_eifs_dur		= OPC_BOOLINT_DISABLED;
	wlan_flags->immediate_xmt		= OPC_BOOLINT_DISABLED;
	wlan_flags->cw_required			= OPC_BOOLINT_DISABLED;
	wlan_flags->nav_updated			= OPC_BOOLINT_DISABLED;
```

```
/* Intialize retry count. */
retry_count = 0;

/* Initialize the packet pointer that holds the last              */
/* transmitted packet to be used for retransmissions when        */
/* necessary.                                                    */
wlan_transmit_frame_copy_ptr = OPC_NIL;

/* Initialize nav duration    */
nav_duration = 0;

/* Initialize receiver idle and conetion window timers.*/
rcv_idle_time = -2.0 * difs_time;
cw_end =        0.0;

/* Initializing the sum of sizes of the packets in the higher layer queue.  */
total_hlpk_size = 0;

/* Initialize the state variables related with the current frame that is being handled.          */
packet_size    = 0;
receive_time   = 0.0;
packet_strm_id = WLANC_STRM_UNSET;

/* Initialize the receiver channel status.                                    */
rcv_channel_status = 0;

/* Data arrived from higher layer is queued in the buffer. Pool memory is used for          */
/* allocating data structure for the higher layer packet and the random destination         */
/* for the packet. This structure is then inserted in the higher layer arrival queue. */
hld_pmh = op_prg_pmo_define ("WLAN hld list elements", sizeof (WlanT_Hld_List_Elem), 32);

/* Obtaining transmitter objid for accessing channel data rate attribute.  */
tx_objid = op_topo_assoc (my_objid, OPC_TOPO_ASSOC_OUT, OPC_OBJTYPE_RATX, 0);

/* If no receiver is attached then generate error message and abort the simulation.          */
if (tx_objid == OPC_OBJID_INVALID)
	{
	wlan_mac_error ("No transmitter attached to this MAC process", OPC_NIL, OPC_NIL);
	}

/* Obtaining number of channels available.  */
op_ima_obj_attr_get (tx_objid, "channel", &chann_objid);
num_chann = op_topo_child_count (chann_objid, OPC_OBJTYPE_RATXCH);

/* Generate error message and terminate simulation if no channel is available for transmission.
*/
if (num_chann == 0)
	{
	wlan_mac_error (" No channel is available for transmission", OPC_NIL, OPC_NIL);
	}

/* Setting the Frequency and Bandwidth for the transmitting channels.  */
for (i = 0; i < num_chann; i++)
```

```
			{
			/* Accessing channel to set the frequency and bandwidth.		*/
			sub_chann_objid = op_topo_child (chann_objid, OPC_OBJTYPE_RATXCH, i);

			/* Setting the operating freqeuncy and channel bandwidth for the transmitting channels.
*/
			op_ima_obj_attr_set (sub_chann_objid, "bandwidth", bandwidth);
			op_ima_obj_attr_set (sub_chann_objid, "min frequency", frequency);
			}

		/* Obtaining receiver's objid for accessing channel data rate attribute.	*/
		rx_objid = op_topo_assoc (my_objid, OPC_TOPO_ASSOC_IN, OPC_OBJTYPE_RARX, 0);

		/* If no receiver is attached then generate error message and abort the simulation.		*/
		if (rx_objid == OPC_OBJID_INVALID)
			{
			wlan_mac_error ("No receiver attached to this MAC process", OPC_NIL, OPC_NIL);
			}

		/* Obtaining number of channels available.	*/
		op_ima_obj_attr_get (rx_objid, "channel", &chann_objid);
		num_chann = op_topo_child_count (chann_objid, OPC_OBJTYPE_RARXCH);

		/* Generate error message and terminate simulation if no channel is available for reception.*/
		if (num_chann == 0)
			{
			wlan_mac_error (" No channel is available for reception", OPC_NIL, OPC_NIL);
			}

		/* Setting the Frequency and Bandwidth for the transmitting channels.	*/
		for (i = 0; i < num_chann; i++)
			{
			/* Accessing channel to set the frequency and bandwidth.		*/
			sub_chann_objid = op_topo_child (chann_objid, OPC_OBJTYPE_RARXCH, i);

			/* Setting the operating freqeuncy and channel bandwidth for the receiving channels.
		*/
			op_ima_obj_attr_set (sub_chann_objid, "bandwidth", bandwidth);
			op_ima_obj_attr_set (sub_chann_objid, "min frequency", frequency);
			}

		llc_iciptr = op_ici_create ("wlan_mac_ind");

		if (llc_iciptr == OPC_NIL)
			{
			wlan_mac_error ("Unable to create ICI for communication with LLC.", OPC_NIL,
OPC_NIL);
			}

		FOUT;
		}

static void
```

```
wlan_higher_layer_data_arrival ()
	{
	Packet*				hld_pkptr;
	int				pk_size, orig_pk_size, stream_id;
	int				i;
	int				dest_addr;
	Ici*				ici_ptr;
	Ici*				strm_info_iciptr;
	Boolean				stn_det_flag;

	/** Queue the packet as it arrives from higher layer.	**/
	/** Also, store the destination address of the packet	**/
	/** in the queue and the arrival time.				**/
	FIN (wlan_higher_layer_data_arrival ());

	/* Get packet from the incomming stream from higher layer and		*/
	/* obtain the packet size						*/
	hld_pkptr = op_pk_get (op_intrpt_strm ());

	/* For bridge and gateway, only accept packet from the higher */
	/* layer if the access point functionality is enabled.		*/
	if (((wlan_flags->gateway_flag == OPC_BOOLINT_ENABLED) ||
		(wlan_flags->bridge_flag == OPC_BOOLINT_ENABLED)) &&
		(ap_flag == OPC_BOOLINT_DISABLED))
		{
		op_pk_destroy (hld_pkptr);
		FOUT;
		}

	pk_size  = op_pk_total_size_get (hld_pkptr);

	/* maintaining total packet size of the packets in the higer layer queue.  */
	total_hlpk_size = total_hlpk_size + pk_size;

	/* If fragmentation is enabled and packet size is greater than the threshold		*/
	/* then MSDU length will not be more than fragmentation threshold, hence		*/
	/* the packet will be fragmented into the size less than or equal to fragmentaion	*/
	/* threshold.										*/
	orig_pk_size = pk_size;
	if ((pk_size > frag_threshold * 8) && (frag_threshold != -1))
		{
		pk_size = frag_threshold * 8;
		}

	/* Destroy packet if it is more than max msdu length or its		*/
	/* size zero. Also, if the size of the higher layer queue		*/
	/* will exceed its maximum after the insertion of this packet,		*/
	/* then discard the arrived packet.					*/
	/* The higher layer is reponsible for the retransmission of		*/
	/* this packet.								*/
	if (pk_size > WLAN_MAXMSDU_LENGTH || pk_size == 0 ||
  total_hlpk_size > hld_max_size)
		{
```

```
/* Write an appropriate simulation log message.                    */
if (drop_pkt_entry_log_flag == OPC_FALSE)
        {
        if (total_hlpk_size > hld_max_size)
                {
                /* Writing log message for dropped packets. */
                op_prg_log_entry_write (drop_pkt_log_handle,
                "SYMPTOMS(S):\n"
          " Wireless LAN MAC layer discarded some packets due to\n "
          " insufficient buffer capacity. \n"
                "\n"
          " This may lead to: \n"
          " - application data loss.\n"
          " - higher layer packet retransmission.\n"
          "\n"
          " REMDEDIAL ACTION (S): \n"
          " 1. Reduce Network laod. \n"
          " 2. User higher speed wireless lan. \n"
          " 3. Increase buffer capacity\n");
                }

        if (pk_size > WLAN_MAXMSDU_LENGTH)
                {
                /* Writing log message for dropped packets due to packet size.*/
                op_prg_log_entry_write (drop_pkt_log_handle,
                "SYMPTOMS(S):\n"
          " Wireless LAN MAC layer discarded some packets due to\n "
          " large packet size. \n"
                "\n"
          " This may lead to: \n"
          " - application data loss.\n"
          " - higher layer packet retransmission.\n"
          "\n"
          " REMDEDIAL ACTION (S): \n"
          " 1. Enable fragmentation threshold. \n"
          " 2. Set the higher layer packet size to \n"
                " be smaller than max MSDU size \n");
                }
        drop_pkt_entry_log_flag = OPC_TRUE;
        }

/* Change the total hold queue size to original value    */
/* as this packet will not be added to the queue.              */
total_hlpk_size = total_hlpk_size - orig_pk_size;

/* Report stat when data packet is dropped due to overflow buffer.        */
op_stat_write (drop_packet_handle, 1.0);
op_stat_write (drop_packet_handle, 0.0);

/* Report stat when data packet is dropped due to overflow buffer.        */
op_stat_write (drop_packet_handle_inbits, (double) (orig_pk_size));
op_stat_write (drop_packet_handle_inbits, 0.0);
op_stat_write (global_dropped_data_handle, (double) (orig_pk_size));
```

```
op_stat_write (global_dropped_data_handle, 0.0);

/* Retrieve the traffic stream information of the packet and	*/
/* update the corresponding per-stream statistics.	*/
strm_info_iciptr = op_pk_ici_get (hld_pkptr);
if ((strm_info_iciptr != OPC_NIL) && (op_ici_attr_exists (strm_info_iciptr,
"stream_id") == OPC_TRUE))
	{
	op_ici_attr_get (strm_info_iciptr, "stream_id", &stream_id);

	/* Register the statistics if this is the first packet we	*/
	/* received belonging to that stream.	*/
	if (stat_reg_status_array [stream_id] == OPC_FALSE)
		{
		wlan_per_stream_stat_register (stream_id);
		}

	/* Update the related per-stream statistics.	*/
	op_stat_write (dropped_data_per_strm_sh_array [stream_id], (double)
(orig_pk_size));
	op_stat_write (dropped_data_per_strm_sh_array [stream_id], 0.0);
	}

/* Destroy the dropped packet.	*/
op_pk_destroy (hld_pkptr);

FOUT;
}

/* Read ICI parameters at the stream interrupt.	*/
ici_ptr = op_intrpt_ici ();

/* Retrieve destination address from the ici set by the interface layer.	*/
if (ici_ptr == OPC_NIL || op_ici_attr_get (ici_ptr, "dest_addr", &dest_addr) ==
OPC_COMPCODE_FAILURE)
	{
	/* Generate error message. */
	wlan_mac_error ("Destination address in not valid.", OPC_NIL, OPC_NIL);
	}

/* If it is a broadcast packet or the station doesn't exist in the subnet	*/
/*if ((dest_addr < 0) || (oms_aa_address_find (oms_aa_wlan_handle, dest_addr) < 0))*/
if (dest_addr < 0)
  {
  /* change the total hld queue size to original value	*/
  /* as this packet will not be added to the queue.	*/
  total_hlpk_size = total_hlpk_size - orig_pk_size;

  op_pk_destroy (hld_pkptr);

  FOUT;
  }
```

```
        /* For an AP bridge, check whether the destination stations exist in the BSS or not.    */
        /* If not, then no need to broadcast the packet.                                       */
        if (wlan_flags->bridge_flag == OPC_BOOLINT_ENABLED && ap_flag ==
OPC_BOOLINT_ENABLED)
                {
                stn_det_flag = OPC_FALSE;
                for (i = 0; i < bss_stn_count; i++)
                        {
                        if (dest_addr == bss_stn_list [i])
                                {
                                stn_det_flag = OPC_TRUE;
                                }
                        }

                /* If the destination station doesn't exist in the BSS then */
                /* no need to broadcast the packet.                        */
                if (stn_det_flag == OPC_FALSE)
                        {
                        /* change the total hld queue size to original value    */
                        /* as this packet will not be added to the queue.       */
                        total_hlpk_size = total_hlpk_size - orig_pk_size;

                        op_pk_destroy (hld_pkptr);

                        FOUT;
                        }
                }

        /* Stamp the packet with the current time. This information will remain        */
        /* unchanged even if the packet is copied for retransmissions, and               */
        /* eventually it will be used by the destination MAC to compute the end-to-      */
        /* end delay.                                                                    */
        op_pk_stamp (hld_pkptr);

        /* Insert the arrived packet in higher layer queue.     */
        wlan_hlpk_enqueue (hld_pkptr, dest_addr);
        FOUT;
        }

static void
wlan_hlpk_enqueue (Packet* hld_pkptr, int dest_addr)
        {
        int                                     list_index;
        char                                    msg_string [120];
        char                                    msg_string1 [120];
        WlanT_Hld_List_Elem*                    hld_ptr;
        double                                  data_size;

        /* Enqueuing data packet for transmission.   */
        FIN (wlan_hlpk_enqueue (Packet* hld_pkptr, int dest_addr));

        /* Allocating pool memory to the higher layer data structure type. */
```

```
		hld_ptr = (WlanT_Hld_List_Elem *) op_prg_pmo_alloc (hld_pmh);

		/* Generate error message and abort simulation if no memory left for data received from higher
	layer.	*/
		if (hld_ptr == OPC_NIL)
			{
			wlan_mac_error ("No more memory left to assign for data received from higher layer",
	OPC_NIL, OPC_NIL);
			}

		/* Updating higher layer data structure fields.		*/
		hld_ptr->time_rcvd		= current_time;
		hld_ptr->destination_address		= dest_addr;
		hld_ptr->pkptr			= hld_pkptr;

		/* Insert a packet to the list.*/
		op_prg_list_insert (hld_list_ptr, hld_ptr, OPC_LISTPOS_TAIL);

		/* Enable the flag indicating that there is a data frame to transmit.		*/
		wlan_flags->data_frame_to_send = OPC_BOOLINT_ENABLED;

		/* Printing out information to ODB.*/
		if (wlan_trace_active == OPC_TRUE)
			{
			sprintf (msg_string, "Just arrived outbound Data packet id %d ", op_pk_id (hld_ptr-
	>pkptr));
			sprintf	(msg_string1, "The outbound Data queue size is %d",		op_prg_list_size
	(hld_list_ptr));
			op_prg_odb_print_major (msg_string, msg_string1, OPC_NIL);
			}

		/* Report stat when outbound data packet is received. */
		op_stat_write (packet_load_handle, 1.0);
		op_stat_write (packet_load_handle, 0.0);

		/* Report stat in bits when outbound data packet is received.	*/
		data_size = (double) op_pk_total_size_get (hld_pkptr);
		op_stat_write (bits_load_handle, data_size);
		op_stat_write (bits_load_handle, 0.0);

		/* Update the global statistics as well.							*/
		op_stat_write (global_load_handle, data_size);
		op_stat_write (global_load_handle, 0.0);

		/* Report outbound data packets queue size at the arrival of every packet.		*/
		op_stat_write (hl_packets_rcvd, (double) (op_prg_list_size (hld_list_ptr)));

		FOUT;
		}

	static void
	wlan_frame_transmit ()
		{
```

```
	char					msg_string  [120];
	char					msg_string1 [120];
	WlanT_Hld_List_Elem*			hld_ptr;
	int					frag_list_size;
	int					type;
	double					pkt_tx_time;
	Ici*					strm_info_iciptr;

	/** Main procedure to call functions for preparing frames.   **/
	/** The procedure to prepare frame is called in this routine **/
	FIN (wlan_frame_transmit());

	/* If Ack and Cts needs to be sent then prepare the appropriate */
	/* frame type for transmission					*/
	if ((fresp_to_send == WlanC_Cts) || (fresp_to_send == WlanC_Ack))
		{
		wlan_prepare_frame_to_send (fresp_to_send);

		/* Break the routine if Cts or Ack is already prepared to tranmsit */
		FOUT;
		}

	/* If it is a retransmission then check which type of frame needs to be	*/
	/* retransmitted and then prepare and transmit that frame				*/
	else if (retry_count != 0)
		{
		/* If the last frame unsuccessfully transmitted was Rts then transmit it again.	*/
		if ((last_frametx_type == WlanC_Rts) && (wlan_flags->rts_sent ==
OPC_BOOLINT_DISABLED))
			{
			/* Retransmit the Rts frame.		*/
			wlan_prepare_frame_to_send (WlanC_Rts);
			}

		/* For the retransmission of data frame first check whether Rts needs to be sent	*/
		/* or not.  If it Rts needs to be sent and it is not already sent then first transmit	*/
		/* Rts and then transmit data frame.						*/
		else if (last_frametx_type == WlanC_Data)
			{
			if ((op_pk_total_size_get (wlan_transmit_frame_copy_ptr) > (8 * rts_threshold
+ WLANC_MSDU_HEADER_SIZE)) &&
				(rts_threshold != -1) && (wlan_flags->rts_sent ==
OPC_BOOLINT_DISABLED))
				{
				/* Retransmit the Rts frame to again contend for the data .	*/
				wlan_prepare_frame_to_send (WlanC_Rts);
				}
			else
				{
				wlan_prepare_frame_to_send (WlanC_Data);
				}
			}
		else
```

```
                    {
                    /* We continue with the retransmission process. We   */
                    /* received the expected Cts for our last Rts.        */
                    /* Hence, now we can retransmit our data frame.       */
                    wlan_prepare_frame_to_send (WlanC_Data);
                    }

            FOUT;
            }

        /* If higher layer queue is not empty then dequeue a packet     */
        /* from the higher layer and insert it into fragmentation       */
        /* buffer check whether fragmentation and Rts-Cts exchange      */
        /* is needed  based on thresholds                               */
        /* Check if fragmenetation buffer is empty. If it is empty      */
        /* then dequeue a packet from the higher layer queue. */
        else if ((op_prg_list_size (hld_list_ptr) != 0) && (op_sar_buf_size (fragmentation_buffer_ptr) ==
0))
            {
            /* If rts is already sent then transmit data otherwise    */
            /* check if rts needs to be sent or not.                  */
            if (wlan_flags->rts_sent == OPC_BOOLINT_DISABLED)
                {
                /* Remove packet from higher layer queue. */
                hld_ptr = (WlanT_Hld_List_Elem*) op_prg_list_remove (hld_list_ptr, 0);

                /* Update the higher layer queue size statistic.        */
                op_stat_write (hl_packets_rcvd, (double) (op_prg_list_size (hld_list_ptr)));

                /* Determine packet size to determine later whether fragmentation      */
                /* and/or rts-cts exchange is needed.                                  */
                packet_size = op_pk_total_size_get (hld_ptr->pkptr);

                /* Updating the total packet size of the higher layer buffer.     */
                total_hlpk_size = total_hlpk_size - packet_size;

                /* Retrieve the traffic stream information if available.          */
                packet_strm_id = WLANC_STRM_UNSET;
                strm_info_iciptr = op_pk_ici_get (hld_ptr->pkptr);
                if (strm_info_iciptr != OPC_NIL && op_ici_attr_exists (strm_info_iciptr,
"stream_id"))
                    {
                    op_ici_attr_get (strm_info_iciptr, "stream_id", &packet_strm_id);
                    }

                /* Setting destination address state variable  */
                destination_addr = hld_ptr->destination_address;

                /* Packet seq number modulo 4096 counter.  */
                packet_seq_number = (packet_seq_number + 1) % 4096;

                /* Packet fragment number is initialized.       */
                packet_frag_number = 0;
```

```
/* Packet needs to be fragmented if it is more          */
/* than fragmentation threshold, provided               */
/* fragmentation is enabled.                            */
if ((packet_size > (frag_threshold * 8)) && (frag_threshold != -1))
        {
        /* Determine number of fragments for the packet        */
        /* and the size of the last fragment                   */

        num_fragments =  (int) (packet_size / (frag_threshold * 8));
        remainder_size = packet_size - (num_fragments * frag_threshold * 8);

        /* If the remainder size is non zero it means that the          */
        /* last fragment is fractional but since the number             */
        /* of fragments is a whole number we need to transmit           */
        /* one additional fragment to ensure that all of the            */
        /* data bits will be transmitted                                */
        if (remainder_size != 0)
                {
                num_fragments = num_fragments + 1;

                }
        }
else
        {
        /* If no fragments needed then number of     */
        /* packets to be transmitted is set to 1     */

        num_fragments = 1;
        remainder_size = packet_size;
        }

/* Storing Data packet id for debugging purposes.       */
pkt_in_service = op_pk_id (hld_ptr->pkptr);

/* Insert packet to fragmentation buffer        */
op_sar_segbuf_pk_insert (fragmentation_buffer_ptr, hld_ptr->pkptr, 0);

/* Computing packet duration in the queue in seconds */
/* and reporting it to the statistics                        */
pkt_tx_time = (current_time - hld_ptr->time_rcvd);

/* Printing out information to ODB.*/
if (wlan_trace_active == OPC_TRUE)
        {
        sprintf (msg_string, "Data packet %d is removed from higher layer
buffer", pkt_in_service);
        sprintf   (msg_string1, "The queueing delay for data packet %d is %fs",

        pkt_in_service, pkt_tx_time);
        op_prg_odb_print_major (msg_string, msg_string1, OPC_NIL);
        }
```

```
				/* Store the arrival time of the packet.		*/
				receive_time = hld_ptr->time_rcvd;

				/* Freeing up allocated memory for the data packet removed from the higher
layer queue.	*/
				op_prg_mem_free (hld_ptr);

				/* Send rts if rts is enabled and packet size is more than rts threshold	*/
				if ((packet_size > (rts_threshold * 8)) && (rts_threshold != -1))
					{
					retry_limit = long_retry_limit;
					/* Prepare Rts frame for transmission		*/
					wlan_prepare_frame_to_send (WlanC_Rts);

	/* Break the routine as Rts is already prepared		*/
					FOUT;
					}
				else
					{
					retry_limit = short_retry_limit;
					}
				}
			}

	/* Prepare data frame to transmit	*/
	wlan_prepare_frame_to_send (WlanC_Data);

	FOUT;
	}

static void
wlan_prepare_frame_to_send (int frame_type)
	{
	char						msg_string [120];
	Packet*						hld_pkptr;
	Packet*						seg_pkptr;
	int						dest_addr, src_addr;
	int						protocol_type = -1;
	int						tx_datapacket_size;
	int						type;
	char						error_string [512];
	int						outstrm_to_phy;
	double						duration, mac_delay;
	WlanT_Data_Header_Fields*			pk_dhstruct_ptr;
	WlanT_Control_Header_Fields*			pk_chstruct_ptr;
	Packet*						wlan_transmit_frame_ptr;

	/*  802.11a Model Addition  */
	/* Add a variable to keep track of the data rate so it can be passed to the pipeline stages. */
	int				rate_holder;

	/*  802.11a Model Addition  */
	/* The control frame transmission rate depends on the given data rate.  */
```

```
/* Adapted from the Philips Lab 802.11a model (dated 11/15/00).   */
double                  control_frame_speed;     /* Speed for control frames. */
int                     next_frag_length;        /* Length of the next fragment (in bits). */
int                     MPDU_size;               /* MPDU length (in bits). */

/** Prepare frames to transmit by setting appropriate fields in the          **/
/** packet format for Data,Cts,Rts or Ack.  If data or Rts packet needs **/
/** to be retransmitted then the older copy of the packet is resent.    **/
FIN (wlan_prepare_frame_to_send (int frame_type));

outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH1;

/*  802.11a Model Addition   */
rate_holder = 1;

/*  802.11a Model Addition   */
/* Compute the control frame speed based on the operational data rate.  */
/* Adapted from the Philips Lab 802.11a model code (dated 11/15/00).  */
control_frame_speed = control_speed (operational_speed);

/* The code is divided as per the frame types */
switch (frame_type)
        {
        case WlanC_Data:
                {
                /*  802.11a Model Addition   */
                /* Base the outgoing data channel on the link speed.  */
                if (operational_speed == 9000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH2;
                        rate_holder = 2;
                        }
                else if (operational_speed == 12000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH3;
                        rate_holder = 3;
                        }
                else if (operational_speed == 18000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH4;
                        rate_holder = 4;
                        }
                else if (operational_speed == 24000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH5;
                        rate_holder = 5;
                        }
                else if (operational_speed == 36000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH6;
                        rate_holder = 6;
                        }
                else if (operational_speed == 48000000)
```

```
                    {
                    outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH7;
                    rate_holder = 7;
                    }
            else if (operational_speed == 54000000)
                    {
                    outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH8;
                    rate_holder = 8;
                    }

            /* If it is a retransmission of a packet then no need      */
            /* of preparing data frame.                           */
            if ((retry_count > 0) && (wlan_transmit_frame_copy_ptr != OPC_NIL))
                    {
                    /* If it is a retransmission then just transmit the previous frame*/
                    wlan_transmit_frame_ptr = op_pk_copy
(wlan_transmit_frame_copy_ptr);

                    /* If retry count is non-zero means that the frame is a */
                    /* retransmission of the last transmitted frame           */
                    op_pk_nfd_access (wlan_transmit_frame_ptr, "Wlan Header",
&pk_dhstruct_ptr);
                    pk_dhstruct_ptr->retry = 1;

                    /* Printing out information to ODB.*/
                    if (wlan_trace_active == OPC_TRUE)
                            {
                            sprintf (msg_string, "Data fragment %d for packet %d is
retransmitted", pk_dhstruct_ptr->fragment_number, pkt_in_service);

                            op_prg_odb_print_major (msg_string, OPC_NIL);
                            }

                    /*  802.11a Model Addition  */
                    /* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).
*/
                    /* Calculate the nav duration that the channel will be occupied by
*/
                    /* the station. The duration is calculated per the 802.11 specification.
*/
                    /* The duration of the ACK frame is determined based on the control
frame  */
                    /* rate.                                              */
                    duration = ppdu_duration (WLAN_ACK_LENGTH,
control_frame_speed) + sifs_time + \ WLAN_AIR_PROPAGATION_TIME + plcp_overhead;

                    /* Since the number of fragments for the last transmitted frame is*/
                    /* already decremented, there will be more fragments to transmit */
                    /* if number of fragments is more than zero.              */
                    if (num_fragments != 1)
                            {
                            /* If more fragments need to be transmitted then the station*/
                            /* needs to broadcast the time until the receipt of the      */
```

```
                                        /* the acknowledgement for the next fragment. 224 bits
(header */
                                        /* size) is the length of the control fields in the data    */
                                        /* frame and needs to be accounted in the duration calculation
      */

                                        /*  802.11a Model Addition  */
                                        /* This situation involves 2 cases: either there are more than */
                                        /* 2 fragments left or exactly two fragments left. If there are */
                                        /* exactly 2 fragments left, then the size of the next fragment */
                                        /* will be the header + remainder size. This result affects the */
                                        /* duration that will be calculated.
                         */
                                        /* Adapted from the Philips Lab 802.11a model (dated
11/15/00)                */
                                        if ((num_fragments > 2) || ((num_fragments == 2) &&
(remainder_size == 0)))
                                              {
                                              next_frag_length =
WLANC_MSDU_HEADER_SIZE + frag_threshold * 8;
                                              }
                                        else if ((num_fragments == 2) && (remainder_size != 0))
                                              {
                                              next_frag_length =
WLANC_MSDU_HEADER_SIZE + remainder_size;
                                              }

                                        /* Use the next_frag_length to recalculate the duration.
      */
                                        duration = 2 * duration + ppdu_duration (next_frag_length,
operational_speed) + \
                                                    sifs_time +
WLAN_AIR_PROPAGATION_TIME + plcp_overhead;
                                        }

                                  /* Station update of it's own nav_duration.    To keep track of the next
      */
                                  /* available contention window.
                                              */
                                  nav_duration = current_time + duration + (double)
(op_pk_total_size_get (wlan_transmit_frame_ptr)) / operational_speed;
                                  }

                         else
                                  {
                                  /*  802.11a Model Addition  */
                                  /* Creating transmit data packet type for use in the 802.11a model. */
                                  wlan_transmit_frame_ptr = op_pk_create_fmt ("wlan_data_802_11a");

                                  /* Prepare data frame fields for transmission.*/
                                  pk_dhstruct_ptr = wlan_mac_pk_dhstruct_create ();
                                  type = WlanC_Data;
```

```
pk_dhstruct_ptr->retry = 0;
pk_dhstruct_ptr->order = 1;
pk_dhstruct_ptr->sequence_number = packet_seq_number;

/*  802.11a Model Addition  */
/* Calculate the nav duration that the channel will be occupied by           */
/* the station. The duration is calculated per the 802.11 specification.
*/
/* The duration of the ACK frame is determined based on the control
frame */
/* rate.
                                                          */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).
*/
duration = ppdu_duration (WLAN_ACK_LENGTH,
control_frame_speed) + sifs_time + \
        WLAN_AIR_PROPAGATION_TIME + plcp_overhead;

/* If there is more than one fragment to transmit and there are  */
/* equal sized fragments then remove fragmentation threshold size
    */
/* length of data from the buffer for transmission.
        */
if  ((num_fragments > 1) || (remainder_size == 0))
    {
    /* Remove next fragment from the fragmentation buffer for
    */
    /* transmission and set the appropriate fragment number.
    */
    seg_pkptr = op_sar_srcbuf_seg_remove
(fragmentation_buffer_ptr, frag_threshold * 8);

    /* Indicate in transmission frame that more fragments need to
be sent  */
    /* if more than one fragments are left
            */
    if (num_fragments != 1)
        {
        pk_dhstruct_ptr->more_frag = 1;

        /* If more fragments need to be transmitted then the
station  */
        /* need to broadcast the time until the receipt of the
    */
        /* the acknowledgement for the next fragment. 224
bits (header      */
        /* size) is the length of control fields in the data frame
    */
        /* and need to be accounted for in the duration
calculation                */

        /*  802.11a Model Addition  */
```

```
                                        /* This situation involves 2 cases: either there are
more than  */
                                        /* 2 fragments left or exactly two fragments left. If
there are */
                                        /* exactly 2 fragments left, then the size of the next
fragment */
                                        /* will be the header + remainder size. This result
affects the */
                                        /* duration that will be calculated.
                              */
                                        /* Adapted from the Philips Lab 802.11a model code
(dated 11/15/00)            */
                                        if ((num_fragments > 2) || ((num_fragments == 2)
&& (remainder_size == 0)))
                                             {
                                             next_frag_length =
WLANC_MSDU_HEADER_SIZE + frag_threshold * 8;
                                             }
                                        else if ((num_fragments == 2) && (remainder_size !=
0))
                                             {
                                             next_frag_length =
WLANC_MSDU_HEADER_SIZE + remainder_size;
                                             }

                                        /* Use the next_frag_length to recalculate the
duration.            */
                                        duration = 2 * duration + ppdu_duration
(next_frag_length, operational_speed) + \
                                             sifs_time +
WLAN_AIR_PROPAGATION_TIME + plcp_overhead;
                                        }
                                   else
                                        {
                                        /* If no more fragments to transmit then set more
fragment field to be 0 */
                                        pk_dhstruct_ptr->more_frag = 0;
                                        }

                                   /* Set fragment number in packet field           */
                                   pk_dhstruct_ptr->fragment_number = packet_frag_number ;

                                   /* Printing out information to ODB. */
                                   if (wlan_trace_active == OPC_TRUE)
                                        {
                                        sprintf (msg_string, "Data fragment %d for packet
%d is transmitted",packet_frag_number, pkt_in_service);

                                        op_prg_odb_print_major (msg_string, OPC_NIL);
                                        }

                                   /* Setting packet fragment number for next fragment to be
transmitted */
```

```
                                        packet_frag_number = packet_frag_number + 1;
                                        }
                                else
                                        {
                                        /* Remove last fragments (if any left) from the fragmentation
buffer for */
                                        /* transmission and disable more fragmentation bit.          */
                                        seg_pkptr = op_sar_srcbuf_seg_remove
(fragmentation_buffer_ptr, remainder_size);

                                        pk_dhstruct_ptr->more_frag = 0;

                                        /* Printing out information to ODB.*/
                                        if (wlan_trace_active == OPC_TRUE)
                                                {
                                                sprintf (msg_string, "Data fragment %d for packet
%d is transmitted",packet_frag_number, pkt_in_service);

                                                op_prg_odb_print_major (msg_string, OPC_NIL);
                                                }

                                        pk_dhstruct_ptr->fragment_number = packet_frag_number;
                                        }

                                /* Setting the Header field structure.          */
                                pk_dhstruct_ptr->duration  = duration;
                                pk_dhstruct_ptr->address1  = destination_addr;
                                pk_dhstruct_ptr->address2  = my_address;

                                /* In the BSS network the Data frame is going from AP to sta then
fromds bit is set.  */
                        if (ap_flag == OPC_BOOLINT_ENABLED)
                                        {
                                        pk_dhstruct_ptr->fromds    = 1;
                                        }
                                else
                                        {
                                        pk_dhstruct_ptr->fromds    = 0;
                                        }

                                /* if in the BSS network the Data frame is going from sta to AP then
tods bit is set.      */
                        if ((bss_flag == OPC_BOOLINT_ENABLED) && (ap_flag ==
OPC_BOOLINT_DISABLED))
                                        {
                                        pk_dhstruct_ptr->tods = 1;

                                        /* If Infrastructure BSS then the immediate destination will be
Access point, which          */
                                        /* then forward the frame to the appropriate destination.
                                                        */
                                        pk_dhstruct_ptr->address1 = bss_id;
                                        pk_dhstruct_ptr->address3 = destination_addr;
```

```
			}
		else
			{
			pk_dhstruct_ptr->tods = 0;
			}

		/* If we are sending the first fragment of the data fragment for the */
		/* first time, then this is the end of media access duration, hence */
		/* we must update the media access delay statistics. */
		if (packet_size == op_pk_total_size_get (seg_pkptr) + op_sar_buf_size
(fragmentation_buffer_ptr))
			{
			mac_delay = current_time - receive_time;
			op_stat_write (media_access_delay, mac_delay);
			op_stat_write (media_access_delay, 0.0);
			op_stat_write (global_mac_delay_handle, mac_delay);
			op_stat_write (global_mac_delay_handle, 0.0);
			}

		op_pk_nfd_set (wlan_transmit_frame_ptr, "Type", type);

		/* Setting the variable which keeps track of the last transmitted frame.
	*/
	 last_frametx_type = type;

		op_pk_nfd_set (wlan_transmit_frame_ptr, "Accept", OPC_TRUE);
		op_pk_nfd_set (wlan_transmit_frame_ptr, "Data Packet ID",
pkt_in_service);

		/* Set the frame control field and nav duration. */
		op_pk_nfd_set (wlan_transmit_frame_ptr, "Wlan Header",
pk_dhstruct_ptr,
		wlan_mac_pk_dhstruct_copy, wlan_mac_pk_dhstruct_destroy, sizeof
(WlanT_Data_Header_Fields));

		/* The actual data is placed in the Frame Body field */
		op_pk_nfd_set (wlan_transmit_frame_ptr, "Frame Body", seg_pkptr);

		/* Make copy of the frame before transmission */
		wlan_transmit_frame_copy_ptr = op_pk_copy
(wlan_transmit_frame_ptr);

		/* 802.11a Model Addition */
		/* Obtain the MSDU size before adding the OFDM PLCP overhead.*/
		/* Adapted from the Philips Lab 802.11a model code (dated 11/15/00).
	*/
		MPDU_size = op_pk_total_size_get (wlan_transmit_frame_ptr);
		op_pk_nfd_set (wlan_transmit_frame_ptr, "MPDU size", MPDU_size);

		/* 802.11a Model Addition */
		/* Include the PLCP overhead in the packet size that will be */
		/* transmitted through the transceiver pipeline. */
```

```
                        /* Adapted from the Philips Lab 802.11a model code (dated 11/15/00).
 */
                        op_pk_total_size_set (wlan_transmit_frame_ptr, \
                            (int) (ppdu_duration (op_pk_total_size_get
(wlan_transmit_frame_ptr), operational_speed) * operational_speed));

                        /* Station update of its own nav_duration     */
                        nav_duration = current_time + duration + (double)
(op_pk_total_size_get (wlan_transmit_frame_ptr)) / operational_speed ;
                        }

                /*  802.11a Model Addition  */
                /* Reporting total number of bits in a data frame.                  */
                /* Note: This reports only the number of bits in the MSDU and does not */
                /* include the overhead associated with the PLCP header and preamble. */
                op_stat_write (data_traffic_sent_handle_inbits, (double) MPDU_size);
                op_stat_write (data_traffic_sent_handle_inbits, 0.0);

                /* If there is nothing in the higher layer data queue and fragmentation   */
                /* buffer then disable the data frame flag which will indicate to the      */
                /* station to wait for the higher layer packet.                 */
                if (op_prg_list_size (hld_list_ptr) == 0 && op_sar_buf_size
(fragmentation_buffer_ptr) == 0)
                        {
                        wlan_flags->data_frame_to_send = OPC_BOOLINT_DISABLED;
                        }

                /* Only expect Acknowledgement for directed frames.          */
                if (destination_addr < 0)
                        {
                        expected_frame_type = WlanC_None;
                        }
                else
                        {
                        /* Ack frame is expected in response to data frame    */
                        expected_frame_type = WlanC_Ack;
                        }

                /* Update data traffic sent stat when the transmission is complete        */
                op_stat_write (data_traffic_sent_handle, 1.0);
                op_stat_write (data_traffic_sent_handle, 0.0);
                break;
                }

        case WlanC_Rts:
                {
                /*  802.11a Model Addition  */
                /* Determine the transmission speed of the RTS frame based on the       */
                /* control frame speed calculated above. The default is already 6 Mbps. */
                if (control_frame_speed == 12000000)
                        {
                        outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH3;
                        rate_holder = 3;
```

```
                }
        else if (control_frame_speed == 24000000)
                {
                outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH5;
                rate_holder = 5;
                }

        /*  802.11a Model Addition   */
        /* Creating RTS packet format type for use in the 802.11a model.  */
        wlan_transmit_frame_ptr = op_pk_create_fmt ("wlan_control_802_11a");

        /* Initializing Rts frame fields        */
        pk_chstruct_ptr = wlan_mac_pk_chstruct_create ();

        /* Type of frame */
        type = WlanC_Rts;

        /* if in the infrastructure BSS network then the immediate receipient for  */
        /* the transmitting station will always be an Access point. Otherwise      */
        /* the frame is directly sent to the final destination.                    */
    if ((bss_flag == OPC_BOOLINT_ENABLED) && (ap_flag ==
OPC_BOOLINT_DISABLED))
                {
                /* If Infrastructure BSS then the immediate destination will be Access
point, which      */
                /* then forward the frame to the appropriate destination.        */
                pk_chstruct_ptr->rx_addr = bss_id;
                }
        else
                {
                /* Otherwise set the final destination address.          */

                pk_chstruct_ptr->rx_addr = destination_addr;
                }

        /* Source station address.  */
        pk_chstruct_ptr->tx_addr = my_address;

        /* Setting the Rts frame type.        */
        op_pk_nfd_set (wlan_transmit_frame_ptr, "Type", type);

        /* Setting the accept field to true, meaning the frame is a good frame.   */
        op_pk_nfd_set (wlan_transmit_frame_ptr, "Accept", OPC_TRUE);

        /* Setting the variable which keeps track of the last transmitted frame that needs
response.        */
  last_frametx_type = type;

        /* Determining the size of the first data fragment or frame that need */
        /* to be transmitted following the Rts transmission.                  */

        if (num_fragments > 1)
                {
```

```
                                /* If there are more than one fragment to transmit then the */
                                /* data segment of the first data frame will be the size of */
                                /* fragmentation threshold. The total packet size will be  */
                                /* data plus the overhead (which is 224 bits).
      */
                                tx_datapacket_size = frag_threshold * 8 +
WLANC_MSDU_HEADER_SIZE;
                                }
                         else
                                /* If there is one data frame to transmit then the        */
                                /* data segment of the first data frame will be the size of */
                                /* the remainder computed earlier. The total packet size   */
                                /* will be data plus the overhead (which is 224 bits).           */
                                {
                                tx_datapacket_size = remainder_size +
WLANC_MSDU_HEADER_SIZE;
                                }

                         /*  802.11a Model Addition  */
                         /* Station is reserving channel bandwidth by using Rts frame, so   */
                         /* in Rts the station will broadcast the duration it needs to send  */

                         /* one data frame and receive ack for it. The total duration is the */
                         /* the time required to transmit one data frame, plus one Cts frame */
                         /* plus one ack frame, plus three sifs interval, and plus        */
                         /* air propagation time for three frames                        */
                         /* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
                         duration = ppdu_duration (WLAN_CTS_LENGTH, control_frame_speed) +
ppdu_duration (WLAN_ACK_LENGTH, control_frame_speed) + \
                                ppdu_duration (tx_datapacket_size, operational_speed) + 3 * (sifs_time
+ WLAN_AIR_PROPAGATION_TIME + plcp_overhead);

                         pk_chstruct_ptr->duration = duration;

                         /* Setting Rts frame fields. */
                         op_pk_nfd_set (wlan_transmit_frame_ptr, "Wlan Header", pk_chstruct_ptr,
wlan_mac_pk_chstruct_copy, wlan_mac_pk_chstruct_destroy, sizeof (WlanT_Control_Header_Fields));


                         /*  802.11a Model Addition  */
                         /* Include PLCP overhead when setting the total size of the RTS packet.
      */
                         op_pk_total_size_set (wlan_transmit_frame_ptr, (int) ((ppdu_duration
(WLAN_RTS_LENGTH, control_frame_speed) + plcp_overhead) * control_frame_speed));

                         /* Station update of its own nav_duration     */
                         nav_duration = current_time + duration + (double) (op_pk_total_size_get
(wlan_transmit_frame_ptr)) / control_frame_speed;

                         /* Cts is expected in response to Rts.         */

                         expected_frame_type = WlanC_Cts;
```

```
			/* Printing out information to ODB.*/
			if (wlan_trace_active == OPC_TRUE)
				{
				sprintf (msg_string, "Rts is being transmitted for data packet %d",
pkt_in_service);
				op_prg_odb_print_major (msg_string, OPC_NIL);
				}

			/* Reporting total number of bits in a control frame.  */
			op_stat_write (ctrl_traffic_sent_handle_inbits, (double)
WLAN_RTS_LENGTH);
			op_stat_write (ctrl_traffic_sent_handle_inbits, 0.0);

			/* Update control traffic sent stat when the transmission is complete	*/
			op_stat_write (ctrl_traffic_sent_handle, 1.0);
			op_stat_write (ctrl_traffic_sent_handle, 0.0);
			break;
			}

		case WlanC_Cts:
			{
			/*  802.11a Model Addition  */
			/* Determine the transmission speed of the CTS frame based on the response */
			/* speed.  The default is already 6 Mbps.					*/
			if (response_speed == 12000000)
				{
				outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH3;
				rate_holder = 3;
				}
			else if (response_speed == 24000000)
				{
				outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH5;
				rate_holder = 5;
				}

			/** Preparing Cts frame in response to the received Rts frame  **/
			/** from the remote station. No response needed for Cts frame.**/

			/*  802.11a Model Addition  */
			/* Creating CTS packet format type for use in the 802.11a model.  */
			wlan_transmit_frame_ptr = op_pk_create_fmt ("wlan_control_802_11a");

			/* Initializing Rts frame fields */
			pk_chstruct_ptr = wlan_mac_pk_chstruct_create ();

			/* Type of frame */
			type = WlanC_Cts;

			/* Destination station address.	*/
			pk_chstruct_ptr->rx_addr = remote_sta_addr;

			/*  802.11a Model Addition  */
			/* Station is reserving channel bandwidth by using Rts frame, so  */
```

```
/* in Rts the station will broadcast the duration it needs to send */

/* one data frame and receive ack for it. The total duration is the */
/* the time required to transmit one Cts frame, plus one data */
/* frame, plus one Ack frame, plus three sifs interval, and plus */
/* three air propagation time for three frames. */
/* In Cts frame the station will transmit the remaining time needed */
/* by the station after the exchange of Rts-Cts */
/* Include the PLCP overhead for the CTS frame. */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00). */
duration = nav_duration - (sifs_time + ppdu_duration (WLAN_CTS_LENGTH,
response_speed) \
        + plcp_overhead + WLAN_AIR_PROPAGATION_TIME +
current_time);
pk_chstruct_ptr->duration = duration;

/* Setting Cts frame type. */
op_pk_nfd_set (wlan_transmit_frame_ptr, "Type", type);

/* Setting the accept field to true, meaning the frame is a good frame. */
op_pk_nfd_set (wlan_transmit_frame_ptr, "Accept", OPC_TRUE);

/* Setting Cts frame fields. */
op_pk_nfd_set (wlan_transmit_frame_ptr, "Wlan Header", pk_chstruct_ptr,
wlan_mac_pk_chstruct_copy,
wlan_mac_pk_chstruct_destroy, sizeof (WlanT_Control_Header_Fields));

/* Setting the total frame size to Cts length. */
/* The PLCP overhead is included in the size. */
op_pk_total_size_set (wlan_transmit_frame_ptr, (int) ((ppdu_duration
(WLAN_CTS_LENGTH, response_speed) + plcp_overhead) * response_speed));

/* Once Cts is transmitted in response to Rts then set the frame */
/* response indicator to none frame as the response is already generated */
fresp_to_send = WlanC_None;


/* No frame is expected once Cts is transmitted */
expected_frame_type = WlanC_None;

/* Printing out information to ODB. */
if (wlan_trace_active == OPC_TRUE)
        {
        sprintf (msg_string, "Cts is being transmitted in response to Rts");
        op_prg_odb_print_major (msg_string, OPC_NIL);
        }

/* Reporting total number of bits in a control frame */
op_stat_write (ctrl_traffic_sent_handle_inbits, (double)
WLAN_CTS_LENGTH);
op_stat_write (ctrl_traffic_sent_handle_inbits, 0.0);

/* Update control traffic sent stat when the transmission is complete */
```

```
			op_stat_write (ctrl_traffic_sent_handle, 1.0);
			op_stat_write (ctrl_traffic_sent_handle, 0.0);
			break;
		}

		case WlanC_Ack:
			{
			/** Preparing acknowledgement frame in response to the data **/
			/** frame received from the remote stations. Note that no     **/
			/** response is needed for the ack frame.                      **/

			/*  802.11a Model Addition  */
			/* Determine the transmission speed of the ACK frame based on the   */
			/* response speed.  The default is already 6 Mbps.                    */
			if (response_speed == 12000000)
				{
				outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH3;
				rate_holder = 3;
				}
			else if (response_speed == 24000000)
				{
				outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH5;
				rate_holder = 5;
				}

			/*  802.11a Model Addition  */
			/* Creating ACK packet format type for the 802.11a model.  */
			wlan_transmit_frame_ptr = op_pk_create_fmt ("wlan_control_802_11a");

			/* Setting ack frame fields  */
			pk_chstruct_ptr = wlan_mac_pk_chstruct_create ();
			type = WlanC_Ack;
			pk_chstruct_ptr->retry = duplicate_entry;

			/*  802.11a Model Addition  */
			/* If there are more fragments to transmit then broadcast the remaining duration
for which          */
			/* the station will be using the channel.              */
			/* Add PLCP overhead to the ACK packet.               */
			/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
			duration = nav_duration - (ppdu_duration (WLAN_ACK_LENGTH,
response_speed) \
				+ plcp_overhead + WLAN_AIR_PROPAGATION_TIME +
current_time);

			pk_chstruct_ptr->duration = duration;

			/* Destination station address.        */
			pk_chstruct_ptr->rx_addr = remote_sta_addr;

			/* Setting Ack type.          */
			op_pk_nfd_set (wlan_transmit_frame_ptr, "Type", type);
```

```
/* Setting the accept field to true, meaning the frame is a good frame.    */
op_pk_nfd_set (wlan_transmit_frame_ptr, "Accept", OPC_TRUE);

op_pk_nfd_set (wlan_transmit_frame_ptr, "Wlan Header", pk_chstruct_ptr,
wlan_mac_pk_chstruct_copy,
wlan_mac_pk_chstruct_destroy, sizeof (WlanT_Control_Header_Fields));

/*  802.11a Model Addition  */
/* Setting the total frame size to Ack length. */
/* The PLCP overhead is included in the size.          */
op_pk_total_size_set (wlan_transmit_frame_ptr, (int) ((ppdu_duration
(WLAN_CTS_LENGTH, response_speed) + plcp_overhead) * response_speed));

/* since no frame is expected,the expected frame type field      */
/* to nil.                                                         */
expected_frame_type = WlanC_None;

/* Once Ack is transmitted in response to Data frame then set the frame  */
/* response indicator to none frame as the response is already generated  */
fresp_to_send = WlanC_None;

/* Printing out information to ODB.*/
if (wlan_trace_active == OPC_TRUE)
        {
        sprintf (msg_string, "Ack is being transmitted for data packet
received");
        op_prg_odb_print_major (msg_string, OPC_NIL);
        }

/* Reporting total number of bits in a control frame.  */
op_stat_write (ctrl_traffic_sent_handle_inbits, (double)
WLAN_ACK_LENGTH);
op_stat_write (ctrl_traffic_sent_handle_inbits, 0.0);

/* Update control traffic sent stat when the transmission is complete*/
op_stat_write (ctrl_traffic_sent_handle, 1.0);
op_stat_write (ctrl_traffic_sent_handle, 0.0);
break;
}

default:
        {
        wlan_mac_error ("Transmission request for unexpected frame type.", OPC_NIL,
OPC_NIL);
        break;
        }
}

/*  802.11a Model Addition  */
/* Before sending the packet to the transmitter, set the Data Rate field  */
/* in the packet header as a way to pass the current link data rate to    */
/* the pipeline stages so the correct OFDM modulation table can be used   */
/* in the dra_ber_11a pipeline stage.                                      */
```

```
	op_pk_nfd_set (wlan_transmit_frame_ptr, "Rate", rate_holder);

	/* Sending packet to the transmitter */
	op_pk_send (wlan_transmit_frame_ptr, outstrm_to_phy);
	wlan_flags->transmitter_busy = OPC_BOOLINT_ENABLED;

	FOUT;
	}

void
wlan_interrupts_process ()
	{
	/** This routine handles the appropriate processing need for each type   **/
	/** of remote interrupt. The type of interrupts are: stream interrupts   **/
	/** (from lower and higher layers), stat interrupts (from receiver and   **/
	/** transmitter).                                                        **/
	FIN (wlan_interrupts_process ());

	/* Check if debugging is enabled.  */
	wlan_trace_active = op_prg_odb_ltrace_active ("wlan");

	/* Determine the current simualtion time        */
	current_time = op_sim_time ();

	/* Determine interrupt type and code to divide treatment        */
	/* along the lines of interrupt type                             */
	intrpt_type = op_intrpt_type ();
	intrpt_code = op_intrpt_code ();

	/* Stream interrupts are either arrivals from the higher layer,  */
	/* or from the physical layer
	*/
	if (intrpt_type == OPC_INTRPT_STRM)
		{
		/* Determine the stream on which the arrival occurred*/
		i_strm = op_intrpt_strm ();

		/* If the event arrived from higher layer then queue the packet */
		/* and the destination address                                   */
		if (i_strm == instrm_from_mac_if)
			{
			/* Process stream interrupt received from higher layer */
			wlan_higher_layer_data_arrival ();
			}

		/* If the event was an arrival from the physical layer,  */
		/* accept the packet and decapsulate it                          */
		else
			{
			/* Process stream interrupt received from physical layer        */

			/*  802.11a Model Addition  */
			/* Capture the data rate of the incoming packet for use in       */
```

```
                /* responding to the data packet.                                    */
                /* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
                switch (i_strm)
                        {
                        case LOW_LAYER_OUT_STREAM_CH1:
                        {
                        response_speed = 6000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH2:
                        {
                        response_speed = 6000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH3:
                        {
                        response_speed = 12000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH4:
                        {
                        response_speed = 12000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH5:
                        {
                        response_speed = 24000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH6:
                        {
                        response_speed = 24000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH7:
                        {
                        response_speed = 24000000;
                        break;
                        }
                        case LOW_LAYER_OUT_STREAM_CH8:
                        {
                        response_speed = 24000000;
                        break;
                        }
                        }

                wlan_physical_layer_data_arrival ();
                }
        }

/* Handle stat interrupt received from the receiver        */
else if (intrpt_type == OPC_INTRPT_STAT)
        {
```

```
		/* Make sure it is not a stat interrupt from the transmitter.	*/
		if (intrpt_code < TRANSMITTER_BUSY_INSTAT)
			{
			/* One of receiver channels is changing its status.		*/
			/* Update the channel status vector.			*/
			wlan_mac_rcv_channel_status_update (intrpt_code);

			/* Update the flag value based on the new status of the		*/
			/* receiver channels.						*/
			if (rcv_channel_status == 0)
				{
				wlan_flags->receiver_busy = OPC_BOOLINT_DISABLED;

				/* Reset the receiver idle timer to the current time since	*/
				/* it became available.						*/
				rcv_idle_time = current_time;
				}
			else
				{
				wlan_flags->receiver_busy = OPC_BOOLINT_ENABLED;
				}
			}
		}

	else if (intrpt_type == OPC_INTRPT_SELF)
		{
		if (intrpt_code == WlanC_CW_Elapsed)
			{
			/* Reset the CW timer, since the period is over, to		*/
			/* enable state transitions.					*/
			cw_end = 0.0;
			}
		}

	FOUT;
	}

static void
wlan_physical_layer_data_arrival ()
	{
	char					msg_string [120];
	int					dest_addr, src_addr;
	int					accept;
	int					data_pkt_id;
	int					final_dest_addr;
	int					rcvd_sta_bssid;
	WlanT_Data_Header_Fields*		pk_dhstruct_ptr;
	WlanT_Control_Header_Fields*		pk_chstruct_ptr;
	WlanT_Mac_Frame_Type			rcvd_frame_type;
	Packet*					wlan_rcvd_frame_ptr;
	Packet*					seg_pkptr;

	/*  802.11a Model Addition  */
```

```
/* Add new variables for the received data rate and the MPDU size of the      */
/* received data packet.                                                       */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00). */
double                                          received_data_rate;
int                                             MPDU_size;

/** Process the frame received from the lower layer.       **/
/** This routine decapsulate the frame and set appropriate  **/
/** flags if the station needs to generate a response to the **/
/** received frame.                                          **/
FIN (wlan_physical_layer_data_arrival ());

/*  Access received packet from the physical layer stream.      */
wlan_rcvd_frame_ptr = op_pk_get (i_strm);

op_pk_nfd_access (wlan_rcvd_frame_ptr, "Accept", &accept);

/* If the packet is received while the station is in transmission          */
/* the packet will not be processed and if needed the station will         */
/* need to retransmit the packet.                                          */
if ((wlan_flags->rcvd_bad_packet == OPC_BOOLINT_ENABLED) || (accept == OPC_FALSE))
        {
        /* If the pipeline stage set the accept flag to be false then it means that  */
        /* the packet is erroneous.  Enable the EIFS duration flag and set           */
        /* nav duration to be EIFS duration.                                         */
        if (accept == OPC_FALSE)
                {
                wlan_flags->wait_eifs_dur = OPC_BOOLINT_ENABLED;

                /* Setting nav duration to EIFS.     */
                nav_duration = current_time + eifs_time;

                /* Reporting the amount of time the channel will be busy.      */
                op_stat_write (channel_reserv_handle, (nav_duration - current_time));
                op_stat_write (channel_reserv_handle, 0.0);
                }

        /* We have experienced a collision during transmission. We              */
        /* could be transmitting a packet which requires a response (an*/
        /* Rts or a data frame requiring an Ack). Even, this is the             */
        /* case, we do not take any action right now and wait for the           */
        /* related timers to expire; then we will retransmit the frame.         */
        /* This is the approach described in the standard, and it is            */
        /* necessary because of the slight possibility that our peer            */
        /* may receive the frame without collision and send us the              */
        /* response back, which we should be still expecting.                   */

        /* Check whether the timer for the expected response has                */
        /* already expired. If yes, we must initiate the retransmission.        */
        if ((expected_frame_type != WlanC_None) && (wlan_flags->transmitter_busy ==
OPC_BOOLINT_DISABLED) &&
                (op_ev_valid (frame_timeout_evh) == OPC_FALSE))
                {
```

```
                    retry_count = retry_count + 1;

                    /* If Rts sent flag was enable then disable it as the station will recontend for the
channel. */
                    if (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED)
                        {
                        wlan_flags->rts_sent = OPC_BOOLINT_DISABLED;
                        }

                    /* Check whether further retries are possible or          */
                    /* the data frame needs to be discarded.                  */
                    wlan_frame_discard ();

                    /* Set expected frame type flag to none as the station needs to retransmit the
frame.    */
                    expected_frame_type = WlanC_None;

                    /* Reset the NAV duration so that the                     */
                    /* retransmission is not unnecessarily delayed.           */
                    nav_duration = current_time;
                    }

                /* No frame response will be generated for bad frame.*/
                fresp_to_send = WlanC_None;

                /* Reset the bad packet receive flag for subsequent receptions. */
                wlan_flags->rcvd_bad_packet = OPC_BOOLINT_DISABLED;

                /* Printing out information to ODB.*/
                if (wlan_trace_active == OPC_TRUE)
                    {
                    sprintf (msg_string, "Received bad packet. Discarding received packet");
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }

                /* Destroy the bad packet.                                */
                op_pk_destroy (wlan_rcvd_frame_ptr);

                /* Break the routine as no further processing is needed. */
                FOUT;
                }

            /* If waiting for EIFS duration then set the nav duration such that    */
            /* the normal operation is resumed.                                    */
            if (wlan_flags->wait_eifs_dur == OPC_BOOLINT_ENABLED)
                {
                nav_duration = current_time;
                wlan_flags->wait_eifs_dur = OPC_BOOLINT_DISABLED;
                }

            /* Getting frame control field and duration information from   */
            /* the received packet.                                         */
            op_pk_nfd_access (wlan_rcvd_frame_ptr, "Type", &rcvd_frame_type) ;
```

```
/* Divide processing based on frame type     */
switch (rcvd_frame_type)
        {
        case WlanC_Data:
                {
                /** First check that wether the station is expecting      **/
                /** any frame or not. If not then decapsulate relevant **/
                /** information from the packet fields and set the        **/
                /** frame response variable with appropriate **/
                /** frame type.                                           **/

                /*  802.11a Model Addition  */
                /* Extract the size of the MPDU from the received data frame and        */
                /* report it.                                                           */
                /* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).  */
                op_pk_nfd_access (wlan_rcvd_frame_ptr, "MPDU size", &MPDU_size);
                op_stat_write (data_traffic_rcvd_handle_inbits, MPDU_size);
                op_stat_write (data_traffic_rcvd_handle_inbits, 0.0);

                /*  Data traffic received report in terms of number of packets.  */
                op_stat_write (data_traffic_rcvd_handle, 1.0);
                op_stat_write (data_traffic_rcvd_handle, 0.0);

                /* Address information, sequence control fields,        */
                /* and the data is extracted from the rcvd packet.      */
                op_pk_nfd_access (wlan_rcvd_frame_ptr, "Wlan Header",
&pk_dhstruct_ptr);

                /* Data packet id of the received data frame is extracted.        */
                op_pk_nfd_access (wlan_rcvd_frame_ptr, "Data Packet ID", &data_pkt_id);

                dest_addr = pk_dhstruct_ptr->address1;
                remote_sta_addr = pk_dhstruct_ptr->address2;

                /* If the station is an AP then it will need to forward the receiving data to this
address. */
                /* Otherwise this field will be zero and will be ignored.        */
                final_dest_addr = pk_dhstruct_ptr->address3;

                fresp_to_send = WlanC_None;

                /* Process frame only if it is destined for this station.  */
                /* Or it is a broadcast frame.                              */
                if ((dest_addr == my_address) || (dest_addr < 0))
                        {
                        /* Extracting the MSDU from the packet only if the packet      */
                        /* is destined for this station.                                */
                        op_pk_nfd_get (wlan_rcvd_frame_ptr, "Frame Body", &seg_pkptr);

                        /* Only send acknowledgement if the data frame is destined for this
station.  */
                        /* No Acks for broadcast frame.                                 */
```

```
if (dest_addr == my_address)
        {
        /* Send the acknowledgement to any received data frame.*/
        fresp_to_send = WlanC_Ack;
        }

/* If its a duplicate packet then destroy it and do nothing */
/* otherwise insert it in the defragmentation list.         */
if (wlan_tuple_find (remote_sta_addr, pk_dhstruct_ptr-
>sequence_number, pk_dhstruct_ptr->fragment_number) == OPC_FALSE)
        {
        wlan_data_process (seg_pkptr, remote_sta_addr,
final_dest_addr, pk_dhstruct_ptr->fragment_number,
                                                pk_dhstruct_ptr-
>more_frag, data_pkt_id, rcvd_sta_bssid);
        }
}
else
        {
        /* Printing out information to ODB.*/
        if (wlan_trace_active == OPC_TRUE)
        {
        sprintf (msg_string, "Data packet %d is received and
discarded", data_pkt_id);
        op_prg_odb_print_major (msg_string, OPC_NIL);
        }

/* If the frame is not destined for this station */
/* then do not respond with any frame.           */
fresp_to_send = WlanC_None;
}

if (expected_frame_type != WlanC_None)
        {
        /* Since the station did not receive the expected frame     */
        /* it has to retransmit the packet.                          */
        retry_count = retry_count + 1;

        /* If Rts sent flag was enable then disable it as the station will recontend
for the channel.   */
        if (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED)
                {
                wlan_flags->rts_sent = OPC_BOOLINT_DISABLED;
                }

        /* Reset the NAV duration so that the                        */
        /* retransmission is not unnecessarily delayed.              */
        nav_duration = current_time;
        }

/* Update nav duration if the received nav duration is greater   */
/* than the current nav duration.                                 */
if (nav_duration < (pk_dhstruct_ptr->duration + current_time))
```

```
                                        {
                                        nav_duration = pk_dhstruct_ptr->duration + current_time;

                                        /* Set the flag that indicates updated NAV value.       */
                                        wlan_flags->nav_updated = OPC_BOOLINT_ENABLED;
                                        }
                                break;
                                }

                case WlanC_Rts:
                                {
                                /** First check that wether the station is expecting any frame or not      **/
                                /** If not then decapsulate the Rts frame and set a Cts frame response    **/
                                /** if frame is destined for this station. Otherwise, just update the       **/
                                /** network allocation vector for this station.                  **/

                                /*  Control Traffic received report in terms of number of bits.            */
                                op_stat_write (ctrl_traffic_rcvd_handle_inbits, (double)
WLAN_RTS_LENGTH);
                                op_stat_write (ctrl_traffic_rcvd_handle_inbits, 0.0);

                                /*  Control Traffic received report in terms of number of packets.         */
                                op_stat_write (ctrl_traffic_rcvd_handle, 1.0);
                                op_stat_write (ctrl_traffic_rcvd_handle, 0.0);

                                op_pk_nfd_access (wlan_rcvd_frame_ptr, "Wlan Header",
        &pk_chstruct_ptr);

                                dest_addr = pk_chstruct_ptr->rx_addr;
                                remote_sta_addr = pk_chstruct_ptr->tx_addr;

                                if (expected_frame_type == WlanC_None)
                                        {
                                        /* We will respond to the Rts with a Cts only if a) the */
                                        /* Rts is destined for us, and b) our NAV duration is           */
                                        /* not larger than current simulation time.                     */
                                        if ((my_address == dest_addr) && (current_time >= nav_duration))
                                                {
                                                /* Set the frame response field to Cts.                 */
                                                fresp_to_send = WlanC_Cts;

                                                /* Printing out information to ODB.          */
                                                if (wlan_trace_active == OPC_TRUE)
                                                        {
                                                        sprintf (msg_string, "Rts is received and Cts will be
transmitted");

                                                        op_prg_odb_print_major (msg_string, OPC_NIL);
                                                        }
                                                }
                                        else
                                                {
                                                /* If Rts is not destined for this station then set the     */
                                                /* frame response field to None                              */
```

```
                        fresp_to_send = WlanC_None;

                        /* Printing out information to ODB.*/
                        if (wlan_trace_active == OPC_TRUE)
                            {
                            sprintf (msg_string, "Rts is received and discarded");
                            op_prg_odb_print_major (msg_string, OPC_NIL);
                            }
                        }
                    }
                else
                    {
                    /* Since the station did not receive the expected frame it has to
retransmit the packet       */
                    retry_count = retry_count + 1;

                    /* If Rts sent flag was enable then disable it as the station will recontend
for the channel.    */
                    if (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED)
                        {
                        wlan_flags->rts_sent = OPC_BOOLINT_DISABLED;
                        }

                    /* Reset the NAV duration so that the                              */
                    /* retransmission is not unnecessarily delayed.                    */
                    nav_duration = current_time;

                    /* Reset the expected frame type variable since we      */
                    /* will retransmit.                         */
                    fresp_to_send = WlanC_None;
                    }

                /* Update nav duration if the received nav duration is greater    */
                /* than the current nav duration.                                 */
                if (nav_duration < (pk_chstruct_ptr->duration + current_time))
                    {
                    nav_duration = pk_chstruct_ptr->duration + current_time;

                    /* Set the flag that indicates updated NAV value.     */
                    wlan_flags->nav_updated = OPC_BOOLINT_ENABLED;
                    }
                break;
                }

            case WlanC_Cts:
                {
                /** First check that whether the station is expecting any frame or not      **/
                /** If not then decapsulate the Rts frame and set a Cts frame response      **/
                /** if frame is destined for this station. Otherwise, just update the        **/
                /** network allocation vector for this station.                  **/

                /*  Control Traffic received report in terms of number of bits.  */
```

```
					op_stat_write (ctrl_traffic_rcvd_handle_inbits, (double)
 WLAN_CTS_LENGTH);
					op_stat_write (ctrl_traffic_rcvd_handle_inbits, 0.0);

					/*  Control Traffic received report in terms of number of packets.		*/
					op_stat_write (ctrl_traffic_rcvd_handle, 1.0);
					op_stat_write (ctrl_traffic_rcvd_handle, 0.0);

					op_pk_nfd_access (wlan_rcvd_frame_ptr, "Wlan Header",
		&pk_chstruct_ptr);
					dest_addr = pk_chstruct_ptr->rx_addr;

					/* If the frame is destined for this station and the station is expecting	*/
					/* Cts frame then set appropriate indicators.				*/
					if ((dest_addr == my_address) && (expected_frame_type == rcvd_frame_type))

						{
						/* The receipt of Cts frame indicates that Rts is successfully	*/
						/* transmitted and the station can now respond with Data frame */
						fresp_to_send = WlanC_Data;

						/* Set the flag indicating that Rts is succesfully transmitted	*/
						wlan_flags->rts_sent = OPC_BOOLINT_ENABLED;

						op_stat_write (retrans_handle, (double) (retry_count * 1.0));
						op_stat_write (retrans_handle, 0.0);

						/* Printing out information to ODB.*/
						if (wlan_trace_active == OPC_TRUE)
							{
							sprintf (msg_string, "Cts is received for Data packet %d",
 pkt_in_service);

							op_prg_odb_print_major (msg_string, OPC_NIL);
							}
						}
					else
						{
						/* Printing out information to ODB.*/
						if (wlan_trace_active == OPC_TRUE)
							{
							sprintf (msg_string, "Cts is received and discarded.");
							op_prg_odb_print_major (msg_string, OPC_NIL);
							}

						/* No response needed as the frame is either not destined for	*/
						/* this station and/or the station is not expecting this frame.	*/
						fresp_to_send = WlanC_None;

						/* Check whether we were expecting another frame. If yes		*/
						/* then we need to retransmit the frame for which we were		*/
						/* expecting a reply.						*/
						if (expected_frame_type != WlanC_None)
							{
```

```
                                        /* Since the station did not receive the expected frame it has to
retransmit the packet      */
                                        retry_count = retry_count + 1;

                                        /* If Rts sent flag was enable then disable it as the station will
recontend for the channel. */
                                        if (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED)
                                                {
                                                wlan_flags->rts_sent =
OPC_BOOLINT_DISABLED;
                                                }

                                        /* Reset the NAV duration so that the                  */
                                        /* retransmission is not unnecessarily delayed.        */
                                        nav_duration = current_time;
                                        }
                                }

                        /* If network allocation vector is less than the received duration    */
                        /* value then update its value.                                        */
                        if (nav_duration < (pk_chstruct_ptr->duration + current_time))
                                {
                                nav_duration = pk_chstruct_ptr->duration + current_time;


                                /* Set the flag that indicates updated NAV value.               */
                                wlan_flags->nav_updated = OPC_BOOLINT_ENABLED;
                                }
                        break;
                        }

                case WlanC_Ack:
                        {
                        /* No response needed for ack frame. */
                        fresp_to_send = WlanC_None;

                        op_pk_nfd_access (wlan_rcvd_frame_ptr,"Wlan Header", &pk_chstruct_ptr);

                        dest_addr = pk_chstruct_ptr->rx_addr;

                        /*  Control Traffic received report in terms of number of bits.        */
                        op_stat_write (ctrl_traffic_rcvd_handle_inbits, (double)
WLAN_ACK_LENGTH);
                        op_stat_write (ctrl_traffic_rcvd_handle_inbits, 0.0);

                        /*  Control Traffic received report in terms of number of packets.     */
                        op_stat_write (ctrl_traffic_rcvd_handle, 1.0);
                        op_stat_write (ctrl_traffic_rcvd_handle, 0.0);

                        if ((dest_addr == my_address) && (rcvd_frame_type == expected_frame_type))
                                {
                                /* Printing out information to ODB.*/
                                if (wlan_trace_active == OPC_TRUE)
```

```
                    {
                    sprintf (msg_string, "Ack received for data packet %d", pkt_in_service);
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }

                op_stat_write (retrans_handle, (double) (retry_count * 1.0));
                op_stat_write (retrans_handle, 0.0);

                /* Reset the retry counter as the expected frame is received      */
                retry_count = 0;

                /* Decrement number of fragment count because one fragment is successfully transmitted.   */
                num_fragments = num_fragments - 1;

                /* When there are no more fragments to transmit then disable the Rts sent flag */
                /* if it was enabled because the contention period due to Rts/Cts exchange is         */
                /* over and another Rts/Cts exchange is needed for next contention period.             */
                if ((num_fragments == 0) && (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED))
                    {
                    wlan_flags->rts_sent = OPC_BOOLINT_DISABLED;

                    /* Set the contention window flag. Since the ACK for the last */
                    /* fragment indicates a sucessful transmission of the entire data,     */
                    /* we need to back-off for a contention window period.    */
                    wlan_flags->cw_required = OPC_TRUE;
                    }

                /* Data packet is successfully delivered to remote station,          */
                /* since no further retransmission is needed the copy of the data */
                /* packet will be destroyed.                                                    */
                op_pk_destroy (wlan_transmit_frame_copy_ptr);
                wlan_transmit_frame_copy_ptr = OPC_NIL;
                }

            else
                {
                /* Printing out information to ODB. */
                if (wlan_trace_active == OPC_TRUE)
                    {
                    sprintf (msg_string, "Ack is received and discarded.");
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }

                /* Check whether we were expecting another frame. If yes then */
                /* we need to retransmit the frame for which we were expecting */
```

```
                                        /* a reply.                                                      */
                                        if (expected_frame_type != WlanC_None)
                                                {
                                                /* Since the station did not receive the expected frame it has to
retransmit the packet       */
                                                retry_count = retry_count + 1;

                                                /* If Rts sent flag was enable then disable it as the station will
recontend for the channel.  */
                                                if (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED)
                                                        {
                                                        wlan_flags->rts_sent =
OPC_BOOLINT_DISABLED;
                                                        }

                                                /* Reset the NAV duration so that the                   */
                                                /* retransmission is not unnecessarily delayed.         */
                                                nav_duration = current_time;
                                                }
                                        }

                                /* If network allocation vector is less than the received duration     */
                                /* value then update its value.                                          */
                                if (nav_duration < (pk_chstruct_ptr->duration + current_time))
                                        {
                                        nav_duration = pk_chstruct_ptr->duration + current_time;

                                        /* Set the flag that indicates updated NAV value.               */
                                        wlan_flags->nav_updated = OPC_BOOLINT_ENABLED;
                                        }
                                break;
                                }

                default:
                                {
                                wlan_mac_error ("Unexpected frame type received.", OPC_NIL, OPC_NIL);
                                break;
                                }
        }

        /* Reporting the amount of time the channel will be busy.       */
        op_stat_write (channel_reserv_handle, (nav_duration - current_time));
        op_stat_write (channel_reserv_handle, 0.0);

        /* Check whether further retries are possible or                */
        /* the data frame needs to be discarded.                        */
        wlan_frame_discard ();

        /* Set the expected frame type to None because either the       */
        /* expected frame is recieved or the station will have to       */
        /* retransmit the frame
        */
        expected_frame_type = WlanC_None;
```

```
	/* Destroying the received frame once relevant information is taken out of it.	*/
	op_pk_destroy (wlan_rcvd_frame_ptr);


	FOUT;
	}

Boolean
wlan_tuple_find (int sta_addr, int seq_id, int frag_num)
	{
	Boolean					result = OPC_BOOLINT_DISABLED;
	int					list_index;
	int					list_size;
	WlanT_Mac_Duplicate_Buffer_Entry*	tuple_ptr;

	/** This routine determines whether the received data frame already exists in the	**/
	/** duplicate buffer. If it is not then it will be added to the list and the list is updated **/
	/** such that its size will will not be greater then the MAX TUPLE SIZE.		**/
	FIN (wlan_tuple_find (sta_addr, seq_id, frag_num));

	/* Finding the index of the station address in the list,			*/
	/* if the station belongs to this subnet.					*/
	list_index = oms_aa_address_find (oms_aa_wlan_handle, sta_addr);

	/* If remote station entry doesn't exist then create new node.	*/
	if (list_index >= 0)
		{
		if (duplicate_list_ptr [list_index] == OPC_NIL)
			{
			/* Creating struct type for duplicate frame (or tuple) structure. */
			tuple_ptr = (WlanT_Mac_Duplicate_Buffer_Entry *)
						op_prg_mem_alloc (sizeof
(WlanT_Mac_Duplicate_Buffer_Entry));

			/* Generate error and abort simulation if no more memory left to allocate for
duplicate buffer	*/
			if (tuple_ptr == OPC_NIL)
				{
				wlan_mac_error ("Cannot allocate memory for duplicate buffer entry",
OPC_NIL, OPC_NIL);
				}

			tuple_ptr->tx_station_address		= remote_sta_addr;

			tuple_ptr->sequence_id			= seq_id;
			tuple_ptr->fragment_number		= frag_num;

			/* Insert new entry in the list.		*/
			duplicate_list_ptr [list_index] = tuple_ptr;

			}
		else
```

```
				{
				if (duplicate_list_ptr [list_index]->sequence_id == seq_id &&
					duplicate_list_ptr [list_index]->fragment_number == frag_num)
					{
					/* This will be set in the retry field of Acknowledgement.	*/
					duplicate_entry = 1;

					/* Break the routine as the packet is already received by the station.*/
					FRET (OPC_TRUE);
					}
				else
					{
					/* Update the sequence id and fragment number fields of the	*/
					/* remote station in the duplicate buffer list. The list		*/
					/* maintains the sequence id and fragment number of the		*/
					/* previously received frame from this remote station.		*/
					duplicate_list_ptr [list_index]->sequence_id = seq_id;
					duplicate_list_ptr [list_index]->fragment_number = frag_num;

					}
				}
	}
		else
			{
			/* Its not possible for a station to directly receive packet from a station that	*/
			/* does not exist in its BSS.
									*/
			wlan_mac_error ("Receiving packet from a station that does not exist in this BSS",
"Possibly wrong destination address", "Please check the configuration");
			}

		/* This will be set in the retry field of Acknowledgement.	*/
		duplicate_entry = 0;

		/* Packet is not already received by the station.	*/
		FRET (OPC_FALSE);
		}


static void
wlan_data_process (Packet* seg_pkptr, int sta_addr, int final_dest_addr, int frag_num, int more_frag, int
pkt_id, int rcvd_sta_bssid)
		{
		char						msg_string [120];
		int						current_index;
		int						list_index;
		int						list_size;
		int						protocol_type;
		WlanT_Mac_Defragmentation_Buffer_Entry*	defrag_ptr;

		/** This routine handles defragmentation process and also sends data to the	**/
		/** higher layer if all the fragments have been received by the station.	**/
```

```
FIN (wlan_data_process (seg_pkptr, sta_addr, final_dest_addr, frag_num, more_frag, pkt_id,
rcvd_sta_bssid));

/* Defragmentation of the received data frame.                              */
/* Inserting fragments into the reassembly buffer. There are      */
/* two possible cases:
        */
/* 1. The remote station has just started sending the              */
/* fragments and it doesn't exist in the list.                        */
/* 2. The remote station does exist in the list and the            */
/* and the new fragment is a series of fragments for the data    */
/* packet.                                                                          */

/* Get the size of the defragmentation list.     */
list_size = op_prg_list_size (defragmentation_list_ptr);

/* Initialize the current node index which will indicate whether */
/* the entry for the station exists in the list.                         */
current_index = -1;

/* Searching through the list to find if the remote station address        */
/* exists i.e. the source station has received fragments for this          */
/* data packet before.
                        */
/* Also, removing entries from the defragmentation buffer which has              */
/* reached its maximum receieve lifetime.                                                */
for (list_index = 0; list_index < list_size; list_index++)
        {
        /* Accessing node of the list for search purposes.      */

        defrag_ptr = (WlanT_Mac_Defragmentation_Buffer_Entry*)
                                                op_prg_list_access (defragmentation_list_ptr,
list_index);

        /* Removing station entry if the receive lifetime has expired.   */
        if ((current_time - defrag_ptr->time_rcvd) >= max_receive_lifetime)
                {
                /* Removing the partially completed fragment once its lifetime has reached.*/
                defrag_ptr =(WlanT_Mac_Defragmentation_Buffer_Entry *)
                                        op_prg_list_remove (defragmentation_list_ptr, list_index);
                op_sar_buf_destroy (defrag_ptr->reassembly_buffer_ptr);

                op_prg_mem_free (defrag_ptr);

                /* Updating the total list size.        */
                list_size = list_size - 1;
                }

        /* If the station entry already exists in the list then store its index for future use.   */
        else if (remote_sta_addr == defrag_ptr->tx_station_address)
                {
                current_index = list_index;
                }
```

```
        }

    /* If remote station entry doesn't exist then create new node     */
    if (current_index == -1)

        {
        /* If the entry of the station does not exist in the defrag list                */
        /* and the fragment received is not the first fragment of the packet            */
        /* then it implies that the maximum receive lifetime of the packet              */
        /* has expired. In this case the received packet will be destroyed and          */
        /* the acknowledgement is sent to the receiver as specified by the              */
        /* protocol.                                                                    */
        if (frag_num > 0)
            {
            op_pk_destroy (seg_pkptr);
            FOUT;
            }

        /* Creating struct type for defragmentation structure   */

        defrag_ptr = (WlanT_Mac_Defragmentation_Buffer_Entry *) op_prg_mem_alloc (sizeof
(WlanT_Mac_Defragmentation_Buffer_Entry));

        /* Generate error and abort simulation if no more memory left to allocate for duplicate
buffer    */
        if (defrag_ptr == OPC_NIL)
            {
            wlan_mac_error ("Cannot allocate memory for defragmentation buffer entry",
OPC_NIL, OPC_NIL);
            }

        /* Source station address is store in the list for future reference.        */
        defrag_ptr->tx_station_address = sta_addr;

        /* For new node creating a reassembly buffer          */
        defrag_ptr->reassembly_buffer_ptr = op_sar_buf_create
(OPC_SAR_BUF_TYPE_REASSEMBLY, OPC_SAR_BUF_OPT_DEFAULT);
        op_prg_list_insert (defragmentation_list_ptr, defrag_ptr, OPC_LISTPOS_TAIL);
        }

  /* Record the received time of this fragment.        */
    defrag_ptr->time_rcvd = current_time;

    /* Insert fragment into the reassembly buffer           */
    op_sar_rsmbuf_seg_insert (defrag_ptr->reassembly_buffer_ptr, seg_pkptr);

    /* If this is the last fragment then send the data to higher layer. */
    if (more_frag == 0)
        {
        /* If no more fragments to rcv then send the data to higher       */
        /* layer and increment rcvd fragment count.                       */
        seg_pkptr = op_sar_rsmbuf_pk_remove (defrag_ptr->reassembly_buffer_ptr);
```

```
if (ap_flag == OPC_BOOLINT_ENABLED)
	{
	/* If the address is not found in the address list then access point will sent the
data to higher	*/
	/* layer for address resolution. Note that if destination address is same as AP's
address then	*/
	/* the packet is sent to higher layer for address resolution.	*/
	if ((oms_aa_address_find (oms_aa_wlan_handle, final_dest_addr) >= 0) &&
(final_dest_addr != my_address))
		{
		/* Printing out information to ODB.*/
		if (wlan_trace_active == OPC_TRUE)
			{
			sprintf (msg_string, "All fragments of Data packet %d is
received and enqueued for transmission within a subnet", pkt_id);
			op_prg_odb_print_major (msg_string, OPC_NIL);
			}

		/* Enqueuing packet for transmission within a subnet. */
		wlan_hlpk_enqueue (seg_pkptr, final_dest_addr);
		}
	else
		{
		/* Update the local/global throughput and end-to-end */
		/* delay statistics based on the packet that will be	*/
		/* forwarded to the higher layer.	*/
		wlan_accepted_frame_stats_update (seg_pkptr);

		/* Set the contents of the LLC-destined ICI -- set the address	*/
		/* of the transmitting station.	*/
		if (op_ici_attr_set (llc_iciptr, "src_addr", remote_sta_addr) ==
OPC_COMPCODE_FAILURE)
			{
			wlan_mac_error ("Unable to set source address in LLC ICI.",
OPC_NIL, OPC_NIL);
			}

		/* Set the destination address (this mainly serves to	*/
		/* distinguish packets received under broadcast conditions.)	*/
		if (op_ici_attr_set (llc_iciptr, "dest_addr", final_dest_addr) ==
OPC_COMPCODE_FAILURE)
			{
			wlan_mac_error("Unable to set destination address in LLC
ICI.", OPC_NIL, OPC_NIL);
			}

		/* Set the protocol type field contained in the Wlan frame.	*/
		protocol_type = 0;
		if (op_ici_attr_set (llc_iciptr, "protocol_type", protocol_type) ==
OPC_COMPCODE_FAILURE)
			{
			wlan_mac_error("Unable to set protocol type in LLC ICI.",
OPC_NIL, OPC_NIL);
```

```
                    }

            /* Printing out information to ODB.*/
            if (wlan_trace_active == OPC_TRUE)
                    {
                    sprintf (msg_string, "All fragments of Data packet %d is
received and sent to the higher layer", pkt_id);
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }

            /* Setting an ici for the higher layer */
            op_ici_install (llc_iciptr);

            /* Sending data to higher layer through mac interface. */
            op_pk_send (seg_pkptr, outstrm_to_mac_if);
            }
        }
    else
        {
        /* If the station is a gateway and not an access point then do not send     */
        /* data to higher layer for address resolution. This is for not allowing  */
        /* data to go out of the Adhoc BSS.                                          */
        if ((wlan_flags->gateway_flag == OPC_BOOLINT_ENABLED) ||
            (wlan_flags->bridge_flag == OPC_BOOLINT_ENABLED))
            {
            /* Printing out information to ODB.*/
            if (wlan_trace_active == OPC_TRUE)
                    {
                    sprintf (msg_string, "Gateway is not an access point so all
received fragments are discarded.");
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }
            op_pk_destroy (seg_pkptr);
            }
        else
            {
            /* Update the local/global throughput and end-to-end */
            /* delay statistics based on the packet that will be       */
            /* forwarded to the higher layer.                          */
            wlan_accepted_frame_stats_update (seg_pkptr);

            /* Printing out information to ODB.*/
            if (wlan_trace_active == OPC_TRUE)
                    {
                    sprintf (msg_string, "All fragments of Data packet %d is
received and sent to the higher layer", pkt_id);
                    op_prg_odb_print_major (msg_string, OPC_NIL);
                    }

            /* Sending data to higher layer through mac interface */
            op_pk_send (seg_pkptr, outstrm_to_mac_if);
            }
        }
```

```
		/* Freeing up memory space once the received data frame is sent to higher layer.  */
		defrag_ptr =(WlanT_Mac_Defragmentation_Buffer_Entry *)
						op_prg_list_remove (defragmentation_list_ptr, current_index);
		op_sar_buf_destroy (defrag_ptr->reassembly_buffer_ptr);

		op_prg_mem_free (defrag_ptr);
		}
	else
		{
		/* Printing out information to ODB.*/
		if (wlan_trace_active == OPC_TRUE)
			{
			sprintf (msg_string, "Data packet %d is received and waiting for more fragments
", pkt_id);
			op_prg_odb_print_major (msg_string, OPC_NIL);
			}
		}

	FOUT;
	}

static void
wlan_accepted_frame_stats_update (Packet* seg_pkptr)
	{
	double		ete_delay, pk_size;
	Ici*		strm_info_iciptr;
	int		stream_id;

	/** This function is called just before a frame received from	**/
	/** physical layer being forwarded to the higher layer to		**/
	/** update end-to-end delay and throughput statistics. **/
	FIN (wlan_accepted_frame_stats_update (seg_pkptr));

	/* Total number of bits sent to higher layer is equivalent to a	*/
	/* throughput.							*/
	pk_size = (double) op_pk_total_size_get (seg_pkptr);
	op_stat_write (throughput_handle, pk_size);
	op_stat_write (throughput_handle, 0.0);

	/* Also update the global WLAN throughput statistic. */
	op_stat_write (global_throughput_handle, pk_size);
	op_stat_write (global_throughput_handle, 0.0);

	/* Compute the end-to-end delay for the frame and record it.	*/
	ete_delay = current_time - op_pk_stamp_time_get (seg_pkptr);
	op_stat_write (ete_delay_handle,		ete_delay);
	op_stat_write (ete_delay_handle,		0.0);
	op_stat_write (global_ete_delay_handle, ete_delay);
	op_stat_write (global_ete_delay_handle, 0.0);

	/* Retrieve the traffic stream information of the packet and	*/
	/* update the corresponding per-stream statistics.		*/
```

```
        strm_info_iciptr = op_pk_ici_get (seg_pkptr);
        if ((strm_info_iciptr != OPC_NIL) && (op_ici_attr_exists (strm_info_iciptr, "stream_id") ==
OPC_TRUE))
                {
                op_ici_attr_get (strm_info_iciptr, "stream_id", &stream_id);

                /* Register the statistics if this is the first packet we     */
                /* received belonging to that stream.                          */
                if (stat_reg_status_array [stream_id] == OPC_FALSE)
                        {
                        wlan_per_stream_stat_register (stream_id);
                        }

                /* Update the related per-stream statistics.                   */
                op_stat_write (ete_delay_per_strm_sh_array [stream_id], ete_delay);
                op_stat_write (ete_delay_per_strm_sh_array [stream_id], 0.0);
                op_stat_write (throughput_per_strm_sh_array [stream_id], pk_size);
                op_stat_write (throughput_per_strm_sh_array [stream_id], 0.0);
                }

        FOUT;
        }

static void
wlan_per_stream_stat_register (int stream_index)
        {
        char                    stat_annot_str [16];

        /** Registers the dimensional per-stream statistics for the given          **/
        /** stream index and updates its status in the status array.               **/
        FIN (wlan_per_stream_stat_register (int stream_index));

        /* Register the statistics at the corresponding dimension.                  */
        ete_delay_per_strm_sh_array [stream_index]    = op_stat_reg ("Wireless LAN Traffic
Stream.Delay (sec)",            stream_index, OPC_STAT_GLOBAL);
        dropped_data_per_strm_sh_array [stream_index] = op_stat_reg ("Wireless LAN Traffic
Stream.Data Dropped (bits/sec)", stream_index, OPC_STAT_GLOBAL);
        throughput_per_strm_sh_array [stream_index]   = op_stat_reg ("Wireless LAN Traffic
Stream.Throughput (bits/sec)",   stream_index, OPC_STAT_GLOBAL);

        /* Annotate the dimensioned statistics to improve their readibility.*/
        sprintf (stat_annot_str, " Stream %d", stream_index);
        op_stat_annotate (ete_delay_per_strm_sh_array [stream_index],    stat_annot_str);
        op_stat_annotate (dropped_data_per_strm_sh_array [stream_index], stat_annot_str);
        op_stat_annotate (throughput_per_strm_sh_array [stream_index],   stat_annot_str);

        /* Update the registration status.                                 */
        stat_reg_status_array [stream_index] = OPC_TRUE;

        FOUT;
        }

static void
```

```
wlan_schedule_deference ()
	{
	/** This routine schedules self interrupt for deference            **/
	/** to avoid collision and also deference to observe               **/
	/** interframe gap between the frame transmission.                 **/
	FIN (wlan_schedule_deference ());

	/* Check the status of the receiver. If it is busy, exit the        */
	/* function, since we will schedule the end of the deference       */
	/* when it becomes idle.                                            */
	if (wlan_flags->receiver_busy == OPC_BOOLINT_ENABLED)
		{
		FOUT;
		}

	/* Extracting current time at each interrupt              */
	current_time = op_sim_time ();

	/* Adjust the NAV if necessary.                           */
	if (nav_duration < rcv_idle_time)
		{
		nav_duration = rcv_idle_time;
		}

	/* Station needs to wait SIFS duration before responding to any           */
	/* frame. Also, if Rts/Cts is enabled then the station needs              */
	/* to wait for SIFS duration after acquiring the channel using            */
	/* Rts/Cts exchange.                                                      */
	if ((fresp_to_send != WlanC_None) || (wlan_flags->rts_sent == OPC_BOOLINT_ENABLED))
		{
		deference_evh = op_intrpt_schedule_self (current_time + sifs_time,
WlanC_Deference_Off);

		/* Disable backoff flag because this frame is a response frame to the      */
		/* previously received frame (this could be Ack or Cts)                   */
		wlan_flags->backoff_flag = OPC_BOOLINT_DISABLED;
		}

	/* If more fragments to send then wait for SIFS duration and transmit.  */
	/* Station need to contend for the channel if one of the fragments is        */
	/* not successfully transmitted.                                             */
	else if ((retry_count == 0) && (op_sar_buf_size (fragmentation_buffer_ptr) > 0))
		{
		/* Scheduling a self interrupt after SIFS duration                         */
		deference_evh = op_intrpt_schedule_self (current_time + sifs_time,
WlanC_Deference_Off);

		/* Disable backoff because the frame need to be transmitted after SIFS duration  */
		/* This frame is part of the fragment burst                                        */
		wlan_flags->backoff_flag = OPC_BOOLINT_DISABLED;
		}
 else
		{
```

```
                /* If the station needs to transmit or retransmit frame, it will          */
                /* defer for nav duration plus  DIFS duration and then backoff            */
                deference_evh = op_intrpt_schedule_self ((nav_duration + difs_time),
WlanC_Deference_Off);

                /* Before sending data frame or Rts backoff is needed.          */
                wlan_flags->backoff_flag = OPC_BOOLINT_ENABLED;
                }

        /* Reset the updated NAV flag, since as of now we scheduled a new        */
        /* "end of deference" interrupt after the last update.                   */
        wlan_flags->nav_updated = OPC_BOOLINT_DISABLED;

        FOUT;
        }

static void
wlan_frame_discard ()
        {
        int seg_bufsize;
        Packet* seg_pkptr;

        /** No further retries for the data frame for which the retry limit has reached.    **/
        /** As a result these frames are discarded.                                         **/
        FIN (wlan_frame_discard ());

        /* If retry limit has reached then drop the frame.        */
        if (retry_count == retry_limit)
                {
                /* Update retransmission count statistic.      */

                op_stat_write (retrans_handle, (double) (retry_count * 1.0));
                op_stat_write (retrans_handle, 0.0);

                /* Update the local and global dropped packet statistics.        */
                op_stat_write (drop_packet_handle, 1.0);
                op_stat_write (drop_packet_handle, 0.0);
                op_stat_write (drop_packet_handle_inbits, (double) packet_size);
                op_stat_write (drop_packet_handle_inbits, 0.0);
                op_stat_write (global_dropped_data_handle, (double) packet_size);
                op_stat_write (global_dropped_data_handle, 0.0);

                /* Also update the per-stream statistics if the packet belongs     */
                /* to a traffic stream.
                        */
                if (packet_strm_id != WLANC_STRM_UNSET)
                        {
                        printf ("I got it\n"); /* IUM */
                        /* Register the statistics if this is the first packet we     */
                        /* received belonging to that stream.                        */
                        if (stat_reg_status_array [packet_strm_id] == OPC_FALSE)
                                {
                                wlan_per_stream_stat_register (packet_strm_id);
```

```
                }

            /* Update the related per-stream statistics.              */
            op_stat_write (dropped_data_per_strm_sh_array [packet_strm_id], packet_size);
            op_stat_write (dropped_data_per_strm_sh_array [packet_strm_id], 0.0);
            }

        /* Reset the retry count for the next packet.   */
        retry_count = 0;

        /* Get the segmenation buffer size to check if there are more fragments left to be
transmitted.    */
        seg_bufsize =  (int) op_sar_buf_size (fragmentation_buffer_ptr);

        if (seg_bufsize != 0)
            {
            /* Discard remaining fragments        */
            seg_pkptr = op_sar_srcbuf_seg_remove (fragmentation_buffer_ptr,
seg_bufsize);

            op_pk_destroy (seg_pkptr);
            }

        /* If expecting Ack frame then destroy the tx data frame as this frame will          */
        /* no longer be transmitted (even if we are not expecting an Ack at this */
        /* moment, we still may have a copy of the frame if at one point in the              */
        /* retransmission history of the original packet we received a Cts for our           */
        /* Rts but then didn't receive an Ack for our data transmission; hence               */
        /* consider this case as well).                                                      */
        if ((expected_frame_type == WlanC_Ack) || (wlan_transmit_frame_copy_ptr !=
OPC_NIL))

            {
            /* Destroy the copy of the frame as the packet is discarded.      */
            op_pk_destroy (wlan_transmit_frame_copy_ptr);
            wlan_transmit_frame_copy_ptr = OPC_NIL;
            }

        /* Reset the flag that indicates successful RTS transmission.              */
        wlan_flags->rts_sent = OPC_BOOLINT_DISABLED;

        /* Reset the "frame to respond" variable unless we have a CTS or          */
        /* ACK to send.                                                           */
        if (fresp_to_send == WlanC_Data)
            {
            fresp_to_send = WlanC_None;
            }

        /* If there is not any other data packet sent from higher layer and       */
        /* waiting in the buffer for transmission, reset the related flag.  */
        if (op_prg_list_size (hld_list_ptr) == 0)
            {
            wlan_flags->data_frame_to_send = OPC_BOOLINT_DISABLED;
            }
        }
```

```
	FOUT;
	}


static void
wlan_mac_rcv_channel_status_update (int channel_id)
	{
	int			mask = 1;

	/** This function updates the status of the receiver's		**/
	/** channel by setting or resetting the corresponding	**/
	/** bit in the rcv_channel_status state variable based		**/
	/** the channel from which the stat interrupt is			**/
	/** received and the value of that channel's statwire.		**/
	FIN (wlan_mac_rcv_channel_status_update (int channel_id));

	/* Create a mask which will access the corresponding */
	/* bit of the channel that is changing its status.		*/
	mask = mask << channel_id;

	/* Set the bit to 1 if channel became busy and to 0 if		*/
	/* the channel became idle without changing the other*/
	/* bits.								*/
	if (op_stat_local_read (channel_id) == 1.0)
		{
		rcv_channel_status = rcv_channel_status | mask;
		}
	else
		{
		rcv_channel_status = rcv_channel_status ^ mask;
		}

	FOUT;
	}

/****** Error handling procedure ******/
static void
wlan_mac_error (char* msg1, char* msg2, char* msg3)
	{

	/** Terminates simulation with an error message.	**/
	FIN (wlan_mac_error (msg1, msg2, msg3));

	op_sim_end ("Error in Wireless LAN MAC process:", msg1, msg2, msg3);

	FOUT;
	}

/*  802.11a Model Addition  */
/* This funcion is called to calculate the rate of transmission of control			*/
/* frames based on the operational data rate provided by the user. The control	*/
/* frame transmission rate is one of 6,12,24 Mbps (i.e. the mandatory data rates		*/
```

```
/* per the 802.11a specification).                                          */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).       */
static double
control_speed (double data_rate)
        {
        FIN (control_speed (double data_rate));

        if ((data_rate == 54E6) || (data_rate == 48E6) || (data_rate == 36E6) || (data_rate == 24E6))
                {
                FRET (24000000);
                }
        else if ((data_rate = 18E6) || (data_rate == 12E6))
                {
                FRET (12000000);
                }
        else
                {
                FRET (6000000);
                }
        FOUT;
        }

/*  802.11a Model Addition  */
/* This function is called to calculate the duration of the data field in a         */
/* given PPDU.  This duratiion includes the PSDU, SERVICE field (16 bits), tail*/
/* bits (6 bits) and enough bit padding to complete the final OFDM symbol.           */
/* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).       */
static double
ppdu_duration (int PSDU_length, double transmission_speed)
        {
        int number_ofdm_symbols;
        FIN (ppdu_duration (int PSDU_length, double transmission_speed));
        number_ofdm_symbols = ceil((16 + 6 + PSDU_length) / (transmission_speed * .000004));
        FRET ((double) number_ofdm_symbols * .000004);
        FOUT;
        }
```

INIT State

```
/***  Enter Executives ***/

/* Initialization of the process model.          */
/* All the attributes are loaded in this routine */
wlan_mac_sv_init ();

/* Schedule a self interrupt to wait for mac interface  */
/* to move to next state after registering              */
op_intrpt_schedule_self (op_sim_time (), 0);


/*** Exit Executives ***/
```

```
/* object id of the surrounding processor      */
my_objid = op_id_self ();

/* Obtain the node's object identifier          */
my_node_objid = op_topo_parent (my_objid);

my_subnet_objid = op_topo_parent (my_node_objid);

/* Obtain the process's process handle          */
own_prohandle = op_pro_self ();

/* Obtain the values assigned to the various attributes */
op_ima_obj_attr_get (my_objid, "Wireless LAN Parameters", &wlan_params_comp_attr_objid);
params_attr_objid = op_topo_child (wlan_params_comp_attr_objid, OPC_OBJTYPE_GENERIC, 0);

/* Obtain the name of the process   */
op_ima_obj_attr_get (my_objid, "process model", proc_model_name);

/* Determine the assigned MAC address which will be used for address resolution.      */
/* Note this is not the final MAC address as there may be static assignments in       */
/* the network.                                                                        */
op_ima_obj_attr_get (my_objid, "station_address", &my_address);

/* Perform auto-addressing for the MAC address. Apart          */
/* from dynamically addressing, if auto-assigned, the          */
/* address resolution function also detects duplicate          */
/* static assignments. The function also initializes          */
/* every MAC address as a valid destination.          */
oms_aa_address_resolve (oms_aa_handle, my_objid, &my_address);

/* Register Wlan MAC process in the model wide registry        */
process_record_handle = (OmsT_Pr_Handle) oms_pr_process_register (
my_node_objid, my_objid, own_prohandle, proc_model_name);

/* If this station is an access point then it has to be registered as an Access Point. */
/* This is because the network will be treated as Infrastructure network once AP is      */
/* detected.                                                                              */
if (ap_flag == OPC_BOOLINT_ENABLED)
        {
        /* Register this protocol attribute and the station address of                   */
        /* this process into the model-wide registry.                                    */
        oms_pr_attr_set (process_record_handle,
                "protocol",        OMSC_PR_STRING,                  "mac",
                "mac_type",        OMSC_PR_STRING,                  "wireless_lan",
                "subprotocol",     OMSC_PR_NUMBER,                  (double) WLAN_AP,
                "subnetid",        OMSC_PR_OBJID,                   my_subnet_objid,
                "address",         OMSC_PR_NUMBER,                  (double) my_address,
                "auto address handle",    OMSC_PR_ADDRESS,          oms_aa_handle,
                OPC_NIL);
        }
else
        {
        /* Register this protocol attribute and the station address of     */
```

```
    /* this process into the model-wide registry.                                    */
    oms_pr_attr_set (process_record_handle,
        "protocol",         OMSC_PR_STRING,             "mac",
        "mac_type",         OMSC_PR_STRING,             "wireless_lan",
        "subprotocol",      OMSC_PR_NUMBER,             (double) WLAN_STA,
        "subnetid",         OMSC_PR_OBJID,              my_subnet_objid,
        "address",          OMSC_PR_NUMBER,             (double) my_address,
        "auto address handle",   OMSC_PR_ADDRESS,       oms_aa_handle,
        OPC_NIL);
    }

/* Obtain the MAC layer information for the local MAC       */
/* process from the model-wide registry.                    */
/* This is to check if the node is a gateway or not.        */
proc_record_handle_list_ptr = op_prg_list_create ();

oms_pr_process_discover (OPC_OBJID_INVALID, proc_record_handle_list_ptr,
    "node objid",      OMSC_PR_OBJID,                   my_node_objid,
    "protocol",        OMSC_PR_STRING,          "bridge",
    OPC_NIL);

/* If the MAC interface process registered itself,              */
/* then there must be a valid match                             */
record_handle_list_size = op_prg_list_size (proc_record_handle_list_ptr);

if (record_handle_list_size != 0)
    {
    wlan_flags->bridge_flag = OPC_BOOLINT_ENABLED;
    }

/* If the station is not a bridge only then check for arp */
if (wlan_flags->bridge_flag == OPC_BOOLINT_DISABLED)
    {
    /* Deallocate memory used for process discovery      */
    while (op_prg_list_size (proc_record_handle_list_ptr))
        {
        op_prg_list_remove (proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);
        }
    op_prg_mem_free (proc_record_handle_list_ptr);

    /* Obtain the MAC layer information for the local MAC             */
    /* process from the model-wide registry.                          */
    proc_record_handle_list_ptr = op_prg_list_create ();

    oms_pr_process_discover (my_objid, proc_record_handle_list_ptr,
        "node objid",      OMSC_PR_OBJID,               my_node_objid,
        "protocol",        OMSC_PR_STRING,              "arp",
        OPC_NIL);

    /* If the MAC interface process registered itself,      */
    /* then there must be a valid match                               */
    record_handle_list_size = op_prg_list_size (proc_record_handle_list_ptr);
    }
```

```
if (record_handle_list_size != 1)
	{
	/* An error should be created if there are more          */
	/* than one WLAN-MAC process in the local node,    */
	/* or if no match is found.                                        */
	wlan_mac_error ("Either zero or several WLAN MAC interface processes found in the node.",
OPC_NIL, OPC_NIL);
	}
else
	{
	/*	Obtain a handle on the process record	*/
	process_record_handle = (OmsT_Pr_Handle) op_prg_list_access (proc_record_handle_list_ptr,
OPC_LISTPOS_HEAD);

	/* Obtain the module objid for the Wlan MAC Interface module	*/
	oms_pr_attr_get (process_record_handle, "module objid", OMSC_PR_OBJID,
&mac_if_module_objid);

	/* Obtain the stream numbers connected to and from the	*/
	/* Wlan MAC Interface layer process	*/
	oms_tan_neighbor_streams_find (my_objid, mac_if_module_objid, &instrm_from_mac_if,
&outstrm_to_mac_if);
	}

/* Deallocate memory used for process discovery	*/
while (op_prg_list_size (proc_record_handle_list_ptr))
	{
	op_prg_list_remove (proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);
	}

op_prg_mem_free (proc_record_handle_list_ptr);

if (wlan_trace_active)
	{
	/* Cache the state name from which this function was */
	/* called.
		*/
	strcpy (current_state_name, "init");
	}
```

BSS_INIT State

```
/*** Enter Executives ***/

/* Schedule a self interrupt to wait for mac interface	*/
/* to move to next state after registering	*/
op_intrpt_schedule_self (op_sim_time (), 0);


/*** Exit Executives ***/
```

```
/* object id of the surrounding processor      */
my_objid = op_id_self ();

/* Obtain the node's object identifier          */
my_node_objid = op_topo_parent (my_objid);
my_subnet_objid = op_topo_parent (my_node_objid);

/* Obtain the values assigned to the various attributes */
op_ima_obj_attr_get (my_objid, "Wireless LAN Parameters", &wlan_params_comp_attr_objid);
params_attr_objid = op_topo_child (wlan_params_comp_attr_objid, OPC_OBJTYPE_GENERIC, 0);

/* Determining the final MAC address after address resolution. */
op_ima_obj_attr_get (my_objid, "station_address", &my_address);

/* Once the station addresses are resolved, then create a pool for wlan addresses. */
oms_aa_address_resolve (oms_aa_wlan_handle, my_objid, &my_address);

/* Obtain the MAC layer information for the local MAC        */
/* process from the model-wide registry.                      */
proc_record_handle_list_ptr = op_prg_list_create ();

oms_pr_process_discover (OPC_OBJID_INVALID, proc_record_handle_list_ptr,
        "subnetid",            OMSC_PR_OBJID,            my_subnet_objid,
        "mac_type",            OMSC_PR_STRING,           "wireless_lan",
        "protocol",            OMSC_PR_STRING,           "mac",
        OPC_NIL);

/* If the MAC interface process registered itself,     */
/* then there must be a valid match                    */
record_handle_list_size = op_prg_list_size (proc_record_handle_list_ptr);

/* Allocating memory for the duplicate buffer based on number of stations in the subnet.  */
duplicate_list_ptr = (WlanT_Mac_Duplicate_Buffer_Entry**)
        op_prg_mem_alloc (record_handle_list_size * sizeof (WlanT_Mac_Duplicate_Buffer_Entry*));

/* Initializing duplicate buffer entries.        */
for (i = 0; i <= (record_handle_list_size - 1); i++)
        {
        duplicate_list_ptr [i] = OPC_NIL;
        }

/* Initialize the address list index to zero.    */
addr_index = 0;

/* Variable to counting number of access point in the network. */
ap_count = 0;

/* Maintain a list of stations in the BSS if it is an AP and a bridge        */
if (ap_flag == OPC_BOOLINT_ENABLED && wlan_flags->bridge_flag ==
OPC_BOOLINT_ENABLED)
        {
        bss_stn_list = op_prg_mem_alloc ((record_handle_list_size - 1) * sizeof (int));
        count = 0;
```

```
        /* Number of stations in the BSS    */
        bss_stn_count = record_handle_list_size - 1;
        }

/* Traversing the process record handle list to determine if there is any access point in the subnet.     */
for (i = 0; i < record_handle_list_size; i++ )
        {
        /*      Obtain a handle on the process record           */
        process_record_handle = (OmsT_Pr_Handle) op_prg_list_access (proc_record_handle_list_ptr, i);

        /* Get the Station type.     */
        oms_pr_attr_get (process_record_handle, "subprotocol", OMSC_PR_NUMBER, &statype);

        /* If the station is an Access Point then its station id will be a BSS id for all the station in that
subnet.  */
        if (statype == (double) WLAN_AP)
                {
                /* If access point found then it means that it is a Infrastructured BSS.    */
                bss_flag = OPC_BOOLINT_ENABLED;

                /* Get the BSS ID.          */
                oms_pr_attr_get (process_record_handle, "address",  OMSC_PR_NUMBER,
&sta_addr);
                bss_id  = (int) sta_addr;

                /* According to IEEE802.11 there cannot be more than one Access point in the same
subnet.  */
                ap_count = ap_count + 1;
                if (ap_count == 2)
                        {
                        wlan_mac_error ("More than one Access Point found.", "Check the
configuration.", OPC_NIL);
                        }
                }

        /* If the station is a bridge and an access point then     */
        /* maintain a list of stations in the BSS                  */
        if (ap_flag == OPC_BOOLINT_ENABLED && wlan_flags->bridge_flag ==
OPC_BOOLINT_ENABLED)
                {
                /* Get the station id          */
                oms_pr_attr_get (process_record_handle, "address",  OMSC_PR_NUMBER,
&sta_addr);

                /* Maintain a list of stations in the BSS not including itself      */
                if ((int) sta_addr != my_address)
                        {
                        bss_stn_list [count] = (int) sta_addr;
                        count = count + 1;
                        }
                }
```

```
        /* Checking the physical characteristic configuration for the subnet.      */
        oms_pr_attr_get (process_record_handle, "module objid", OMSC_PR_OBJID, &my_objid);

        /* Obtain the values assigned to the various attributes */
        op_ima_obj_attr_get (my_objid, "Wireless LAN Parameters", &wlan_params_comp_attr_objid);
        params_attr_objid = op_topo_child (wlan_params_comp_attr_objid, OPC_OBJTYPE_GENERIC,
0);

        /* Load the appropriate physical layer characteristics. */
        op_ima_obj_attr_get (params_attr_objid, "Physical Characteristics", &sta_phy_char_flag);

        if (sta_phy_char_flag != phy_char_flag)
                {
                wlan_mac_error ("Physical Characteristic configuration mismatch in the subnet.",
                                                    "All stations in the subnet should have same physical
characteristics", "Check the configuration");
                }
  }

/* Deallocate memory used for process discovery      */
while (op_prg_list_size (proc_record_handle_list_ptr))
        {
        op_prg_list_remove (proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);
        }
op_prg_mem_free (proc_record_handle_list_ptr);

/* Obtain the MAC layer information for the local MAC        */
/* process from the model-wide registry.                     */
/* This is to check if the node is a gateway or not.         */
proc_record_handle_list_ptr = op_prg_list_create ();

oms_pr_process_discover (OPC_OBJID_INVALID, proc_record_handle_list_ptr,
        "node objid",           OMSC_PR_OBJID,                          my_node_objid,
        "gateway node",         OMSC_PR_STRING,                         "gateway",
        OPC_NIL);

/* If the MAC interface process registered itself,      */
/* then there must be a valid match                     */
record_handle_list_size = op_prg_list_size (proc_record_handle_list_ptr);

if (record_handle_list_size != 0)
        {
        wlan_flags->gateway_flag = OPC_BOOLINT_ENABLED;
        }

/* Deallocate memory used for process discovery      */
while (op_prg_list_size (proc_record_handle_list_ptr))
        {
        op_prg_list_remove (proc_record_handle_list_ptr, OPC_LISTPOS_HEAD);
        }
op_prg_mem_free (proc_record_handle_list_ptr);
```

IDLE State

```
/*** Enter Executives ***/

/** The purpose of this state is to wait until the packet has            **/
/** arrived from the higher or lower layer.                              **/
/** In this state following intrpts can occur:                           **/
/** 1. Data arrival from application layer                               **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer                      **/
/** 3. Busy intrpt stating that frame is being rcvd                      **/
/** 4. Coll intrpt indicating that more than one frame is rcvd      **/
/* When Data arrives from the application layer, insert it               */
/* in the queue.                                                         */
/* If rcvr is not busy then   set Deference to DIFS                      */
/* and Change state to "DEFER" state                                     */

/* Rcvd RTS,CTS,DATA,or ACK (frame rcvd intrpt)                     */
/* Set Backoff flag if the station needs to backoff                      */
/* If the frame is destined for this station then send                   */
/* appropriate response and set deference to SIFS                        */
/* clear the rcvr busy flag and clamp any data transmission              */
/*                                                                       */
/* If it's a broadcast frame then set deference to NAV                   */
/* and schedule self intrpt and change state to "DEFER".                 */
/* Copy the frame (RTS/DATA) in retransmission variable                  */
/* if rcvr start receiving frame (busy stat intrpt) then set             */
/* a flag indicating rcvr is busy,if rcvr start receiving                */
/* more than one frame (collision stat intrpt) then set the              */
/* rcvd frame as invalid frame set deference time to EIFS                */

if (wlan_trace_active)
        {
        /* Determine the current state name.          */
        strcpy (current_state_name, "idle");
        }


/*** Exit Executives ***/

/* Interrupt processing routine        */
wlan_interrupts_process ();

/* Schedule deference interrupt when there is a frame to transmit        */
/* at the stream interrupt and the receiver is not busy                  */
if (READY_TO_TRANSMIT)
        {
        /* If the medium was idling for a period equal or longer than    */
        /* DIFS time then we don't need to defer.                        */
        if (MEDIUM_IS_IDLE)
                {
                /* We can start the transmission immediately.            */
                wlan_flags->immediate_xmt = OPC_TRUE;
                backoff_slots = 0;
```

```
                }
        else
                {
                /* We need to defer. Schedule the end of it.              */
                wlan_schedule_deference ();
                }

        /* If we are in the contention window period, cancel the self   */
        /* interrupt that indicates the end of it. We will reschedule    */
        /* if it will be necessary.
        */
        if (intrpt_type == OPC_INTRPT_STRM && op_ev_valid (cw_end_evh) == OPC_TRUE)
                {
                op_ev_cancel (cw_end_evh);
                }
        }
```

DEFER State

```
/*** Enter Executives ***/

/** This state defer until the medium is available for transmission      **/
/** Interrupts that can occur in this state are:                          **/
/** 1. Data arrival from application layer                                **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer                       **/
/** 3. Busy intrpt stating that frame is being rcvd                       **/
/** 4. Collision intrpt stating that more than one frame is rcvd          **/
/** 5. Deference timer has expired (self intrpt)                          **/
/** For Data arrival from application layer queue the packet.             **/
/** Set Backoff flag if the station needs to backoff                      **/
/** after deference because the medium is busy                            **/
/** If the frame is destined for this station then set                    **/
/** frame to respond and set a deference timer to SIFS.                   **/
/** Set deference timer to SIFS and don't change states                   **/
/** If rcvr start receiving more than one frame then flag the             **/
/** rcvd frame as invalid frame and set a deference to EIFS.              **/

if (wlan_trace_active)
        {
        /* Determine the current state name.          */
        strcpy (current_state_name, "defer");
        }


/*** Exit Executives ***/

/* Call the interrupt processing routine for each interrupt        */
wlan_interrupts_process ();

/* If the receiver is busy while the station is deferring */
/* then clear the self interrupt. As there will be a new self      */
/* interrupt generated once the receiver becomes idle again.       */
```

```
if (RECEIVER_BUSY_HIGH  && (op_ev_valid (deference_evh) == OPC_TRUE))
        {
        op_ev_cancel (deference_evh);
        }

/* If the receiver becomes idle again schedule the end of the     */
/* deference.                                                     */
if (RECEIVER_BUSY_LOW)
        {
        wlan_schedule_deference ();
        }

/* While we were deferring, if we receive a frame which           */
/* requires a response, then we need to re-schedule our end of    */
/* deference interrupt, since the deference time for response     */
/* frames is shorter. Similarly, we need to re-schedule it if     */
/* the received frame made us set our NAV to a higher value.      */
else if (FRAME_RCVD && (fresp_to_send != WlanC_None || wlan_flags->nav_updated ==
OPC_BOOLINT_ENABLED) &&
        (op_ev_valid (deference_evh) == OPC_TRUE))
        {
        /* Cancel the current event and schedule a new one.       */
        op_ev_cancel (deference_evh);
        wlan_schedule_deference ();
        }
```

BKOFF_NEEDED State

```
/*** Enter Executives ***/

/** Determining wether to backoff. It is needed when station preparing   **/
/** to transmit frame discovers that the medium is busy or when the      **/
/** the station infers collision.                                        **/
/** Backoff is not needed when the station is responding to the frame.   **/
/** If backoff needed then check wether the station completed its        **/
/** backoff in the last attempt. If not then resume the backoff          **/
/** from the same point, otherwise generate a new random number          **/
/** for the number of backoff slots.                                     **/

/* Checking wether backoff is needed or not */
if (wlan_flags->backoff_flag == OPC_BOOLINT_ENABLED)
        {
        if (backoff_slots == 0)
                {
                /* Compute backoff interval using binary exponential process */
                if (retry_count != 0)
                        {
                        /* Set the maximum backoff for the uniform distribution     */
                        max_backoff = max_backoff * 2 + 1;
                        }
                else
                        {
```

```
                    /* if retry count is set to 0 then set the              */
                    /* maximum backoff slots to min window size             */
                    max_backoff = cw_min;
                    }

                /* The number of possible slots grows exponentially         */
                /* until it exceeds a fixed limit.                          */
                if (max_backoff > cw_max)
                    {
                    max_backoff = cw_max;
                    }

                /* Obtain a uniformly distributed random integer between 0 and the minimum contention
window size     */
                /* Scale the number of slots according to the number of retransmissions.*/
                backoff_slots = floor (op_dist_uniform (max_backoff + 1));
                }

        /* Set a timer for the end of the backoff interval.   */
        intrpt_time = (current_time + backoff_slots * slot_time);

        /* Scheduling self interrupt for backoff */
        backoff_elapsed_evh = op_intrpt_schedule_self (intrpt_time, WlanC_Backoff_Elapsed);

        /* Reporting number of backoff slots as a statistic */
        op_stat_write (backoff_slots_handle, backoff_slots);
        op_stat_write (backoff_slots_handle, 0.0);
        }
```

BACKOFF State

```
/*** Enter Executives ***/

/** Processing Random Backoff                                   **/
/** In this state following intrpts can occur:          **/
/** 1. Data arrival from application layer                      **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer             **/
/** 3. Busy intrpt stating that frame is being rcvd             **/
/** 4. Coll intrpt stating that more than one frame is rcvd     **/
/** Queue the packet for Data Arrival from application          **/
/** layer and do not change the state.                          **/
/** If the frame is destined for this station then prepare      **/
/** appropriate frame to respond and set deference to SIFS      **/
/** Update NAV value (if needed) and reschedule deference       **/
/** Change state to "DEFER"                                     **/
/** If it's a broadcast frame then check wether NAV needs       **/
/** to be updated. Schedule self interrupt and change           **/
/** state to Deference                                          **/
/** If rcvr start receiving frame (busy stat intrpt) then **/
/** set a flag indicating rcvr is busy.                         **/
/** if rcvr start receiving more than one frame then flag       **/
/** the rcvd frame as invalid and set deference                 **/
```

```
/** timer to EIFS                                       **/

/* Change State to DEFER                          */
if (wlan_trace_active)
        {
        /* Determine the current state name.      */
        strcpy (current_state_name, "backoff");
        }

/*** Exit Executives ***/

/* Call the interrupt processing routine for each interrupt       */
wlan_interrupts_process ();

/* Set the number of slots to zero, once the backoff is completed          */
if (BACKOFF_COMPLETED)
        {
        backoff_slots = 0.0;
        }

/* Storing remaining backoff slots if the frame is rcvd from the remote station*/
if (RECEIVER_BUSY_HIGH)
        {
        /* Computing remaining backoff slots for next iteration */
        backoff_slots =  ceil ((intrpt_time - current_time) / slot_time);

        if (op_ev_valid (backoff_elapsed_evh) == OPC_TRUE)
                {
                /* clear the self interrupt as station needs to defer */
                op_ev_cancel (backoff_elapsed_evh);
                }
        }

/* Schedule deference if the frame is received while the station is backing off.*/
if (FRAME_RCVD)
        {
        wlan_schedule_deference ();
        }
```

TRANSMIT State

```
/***  Enter Executives ***/

/** In this state following intrpts can occur:                      **/
/** 1. Data arrival from application layer.                         **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer.                **/
/** 3. Busy intrpt stating that frame is being rcvd.                **/
/** 4. Collision intrpt means more than one frame is rcvd.          **/
/** 5. Transmission completed intrpt from physical layer            **/
/** Queue the packe for Data Arrival from the higher layer,         **/
/** and do not change state.                                        **/
/** After Transmission is completed change state to FRM_END         **/
```

```
/** No response is generated for any lower layer packet arrival **/

/* Prepare transmission frame by setting appropriate          */
/* fields in the control/data frame.                          */
/* Skip this routine if any frame is received from the        */
/* higher or lower layer(s)                                   */
if (wlan_flags->immediate_xmt == OPC_TRUE)
        {
        wlan_frame_transmit ();
        wlan_flags->immediate_xmt = OPC_FALSE;
        }

else if (wlan_flags->rcvd_bad_packet == OPC_BOOLINT_DISABLED &&
   intrpt_type == OPC_INTRPT_SELF)
        {
        wlan_frame_transmit ();
        }

if (wlan_trace_active)
        {
        /* Determine the current state name */
        strcpy (current_state_name, "transmit");
        }

/*** Exit Executives ***/

/* If the packet is received while the the station is       */
/* transmitting then mark the received packet as bad.       */
if (op_intrpt_type () == OPC_INTRPT_STAT)
        {
        intrpt_code = op_intrpt_stat ();
        if (intrpt_code < TRANSMITTER_BUSY_INSTAT && op_stat_local_read (intrpt_code) == 1.0
&& rcv_channel_status == 0)
                {
                wlan_flags->rcvd_bad_packet = OPC_BOOLINT_ENABLED;
                }

        /* If we completed the transmission then reset the          */
        /* transmitter flag.                                        */
        else if (intrpt_code == TRANSMITTER_BUSY_INSTAT)
                {
                wlan_flags->transmitter_busy = OPC_BOOLINT_DISABLED;

                /* Also reset the receiver idle time, since with    */
                /* the end of our transmission, we expect that the  */
                /* medium became idle again (but make sure we are   */
                /* also not receiving a packet).                    */
                if (rcv_channel_status == 0)
                        {
                        rcv_idle_time = op_sim_time ();
                        }
                }
        }
```

```
else if ((op_intrpt_type () == OPC_INTRPT_STRM) && (op_intrpt_strm () != instrm_from_mac_if))
        {
        /* While transmitting, we received a packet from          */
        /* physical layer. Mark the packet as bad.                */
        wlan_flags->rcvd_bad_packet = OPC_BOOLINT_ENABLED;
        }

/* Call the interrupt processing routine for each interrupt.*/
wlan_interrupts_process ();
```

FRM_END State

```
/*** Enter Executives ***/

/** The purpose of this state is to determine the next unforced        **/
/** state after completing transmission.                                **/

/** 3 cases                                                             **/
/** 1.If just transmitted RTS or DATA frame then wait for              **/
/** response with expected_frame_type variable set and change          **/
/** the states to Wait for Response otherwise just DEFER for           **/
/** next transmission                                                  **/
/** 2.If expected frame is rcvd then check to see what is the          **/
/** next frame to transmit and set appropriate deference timer **/
/** 2a.If all the data fragments are transmitted then check            **/
/** wether the queue is empty or not                         **/
/** If not then based on threshold fragment the packet                 **/
/** and based on threshold decide wether to send RTS or not **/
/** If there is a data to be transmitted then wait for DIFS            **/
/**        duration before contending for the channel                  **/
/** If nothing to transmit then go to IDLE state             **/
/** and wait for the packet arrival from higher or lower layer         **/
/** 3.If expected frame is not rcvd then infer collision,              **/
/** set backoff flag, if retry limit is not reached                    **/
/** retransmit the frame by contending for the channel                 **/

/* If there is no frame expected then check to see if there            */
/* is any other frame to transmit. Also mark the channel as idle */
if (expected_frame_type == WlanC_None)
        {
        /* If the frame needs to be retransmitted or there is          */
        /* something in the fragmentation buffer to transmit or the    */
        /* station needs to respond to a frame then schedule           */
        /* deference.                                                  */
        if (op_sar_buf_size (fragmentation_buffer_ptr) != 0 || retry_count != 0 || fresp_to_send !=
WlanC_None)
                {
                /* Schedule deference before frame transmission        */
                wlan_schedule_deference ();
                }
```

```
/* After completing a successful frame transmission, even       */
/* though we don't have any other frame to transmit, still      */
/* we need to execute to backoff algorithm to generate a        */
/* contention window period and back-off during that period     */
/* as stated in the protocol.                               */
else if (wlan_flags->cw_required == OPC_TRUE)
        {
        /* Determine the size of the contentions window.        */
        cw_slots = floor (op_dist_uniform (cw_min + 1));
        cw_end = current_time + difs_time +  cw_slots * slot_time;

        /* Schedule a self interrupt indicating the end of the  */
        /* contention window.                                   */
        cw_end_evh = op_intrpt_schedule_self (cw_end, WlanC_CW_Elapsed);

        /* Update the backoff time statistic.           */
        op_stat_write (backoff_slots_handle, cw_slots);
        op_stat_write (backoff_slots_handle, 0.0);

        /* Reset the flag since we scheduled the period.        */
        wlan_flags->cw_required = OPC_FALSE;
        }

else if (cw_end > current_time)
        {
        /* We are in the contention window period, but we had          */
        /* to leave the "idle" state to send a response (Cts,          */
        /* Ack) for a frame we received. Now we are moving back        */
        /* to idle state. Hence, re-schedule the self interrupt        */
        /* that will indicate the end of the contention window. */
        cw_end_evh = op_intrpt_schedule_self (cw_end, WlanC_CW_Elapsed);
        }

else
        {
        /* Schedule the deference if we have a frame in the            */
        /* buffer sent from higher layer for transmission,             */
        /* since the contention window period is over.                 */
        if (op_prg_list_size (hld_list_ptr) != 0)
                {
                /* Schedule deference before frame transmission        */
                wlan_schedule_deference ();
                }

        /* Reset the end of the CW timer, since it is over.            */
        cw_end = 0.0;
        }
}
else
    {
    /*  802.11a Model Addition                                  */
    /* The station needs to wait for the expected frame type    */
    /* So it will set the frame timeout interrupt which will be */
```

```
/* exectued if no frame is received in the set duration.                    */
        /* Adapted from the Philips Labs 802.11a model code (dated 11/15/00).        */
        timer_duration = WLAN_ACK_LENGTH / control_speed (operational_speed) + sifs_time +
plcp_overhead + WLAN_AIR_PROPAGATION_TIME;
        frame_timeout_evh = op_intrpt_schedule_self (current_time + timer_duration,
WlanC_Frame_Timeout);
        }
```

WAIT_FOR_RESPONSE State

```
/*** Enter Executives ***/

/** The purpose of this state is to wait for the response after              **/
/** transmission. The only frames which require acknowlegements              **/
/**  are RTS and DATA frame.                                                  **/
/** In this state following intrpts can occur:                                **/
/** 1. Data arrival from application layer                                    **/
/** 2. Frame (DATA,ACK,RTS,CTS) rcvd from PHY layer                           **/
/** 3. Frame timeout if expected frame is not rcvd                            **/
/** 4. Busy intrpt stating that frame is being rcvd                           **/
/** 5. Collision intrpt stating that more than one frame is rcvd **/
/** Queue the packet as Data Arrives from application layer                   **/
/** If Rcvd unexpected frame then collision is inferred and                   **/
/** retry count is incremented                                                **/
/** if a collision stat interrupt from the rcvr then flag the                 **/
/** received frame as bad                                                     **/

if (wlan_trace_active)
        {
        /* Determine the current state name.          */
        strcpy (current_state_name, "wait_for_response");
        }

/*** Exit Executives ***/

/* Clear the frame timeout interrupt once the receiver is busy           */
/* or the frame is received (in case of collisions, the                  */
/* frames whose reception has started while we were                      */
/* transmitting are excluded in the FRAME_RCVD macro).                   */
intrpt_type = op_intrpt_type ();
if (((intrpt_type == OPC_INTRPT_STAT && op_intrpt_stat () < TRANSMITTER_BUSY_INSTAT &&
            op_stat_local_read (op_intrpt_stat ()) == 1.0 && rcv_channel_status == 0) ||
           FRAME_RCVD) &&
        (op_ev_valid (frame_timeout_evh) == OPC_TRUE))
        {
        op_ev_cancel (frame_timeout_evh);
        }

/* Call the interrupt processing routine for each interrupt       */
/* request.                                                        */
wlan_interrupts_process ();
```

```
/* If expected frame is not received in the set          */
/* duration or the there is a collision at the           */
/* receiver then set the expected frame type to          */
/* be none because the station needs to retransmit       */
/* the frame.                                            */
if (FRAME_TIMEOUT)
        {
        /* Setting expected frame type to none frame */
        expected_frame_type = WlanC_None;

        /* retransmission counter will be incremented */
        retry_count = retry_count + 1;

        /* Reset the NAV duration so that the                  */
        /* retransmission is not unnecessarily delayed.        */
        nav_duration = current_time;

        /* Check whether further retries are possible or       */
        /* the data frame needs to be discarded.               */
        wlan_frame_discard ();
        }
```

APPENDIX B. SNR-BASED RATE AGILITY OPNET CODE

This appendix contains the modifications to the *wlan_mac_11a* process model used to realize the SNR-based dynamic data rate agility mechanism presented in Chapter IV. With the exception of the *max_operational_speed* variable initialization in the *wlan_mac_svc_init* function and the addition of the data rate statistic collection function, the only code additions required to implement the SNR-based rate agility mechanism are to the *wlan_physical_layer_data_arrival* function. The block of code presented below was added to *wlan_physical_layer_data_arrival* immediately after the code used to obtain the frame control field and duration information from the arriving packet (i.e., line 1667 of the *wlan_mac_11a* function block). Comments indicating the nature of the code changes are included.

OPNET Code additions to *wlan_physical_layer_data_arrival* Bryan E. Braswell March 2001

```
/* SNR-Based Data Rate Agility Addition.                          */
/* Access the SNR from the received packet and use the SNR to adjust  */
/* the data rate based on the maximum speed as defined by the user.   */
op_pk_nfd_access (wlan_rcvd_frame_ptr, "Link SNR", &snr_holder);

/* SNR-Based Data Rate Agility Addition.  */

/* This structure is used to Compare the received SNR to the    */
/* thresholds to determine the new data rate. The new data           */
/* rate can only be as high as the user-defined max data rate        */
/* obtained at the start of the simulation.                     */
if (max_operational_speed == 54000000)
        {
        if (snr_holder >= 12.22)
                {
                new_operational_speed = max_operational_speed;
                }
        else if ((snr_holder >= 9.70) && (snr_holder < 12.22))
                {
                new_operational_speed = 48000000;
                }
        else if ((snr_holder >= 8.86) && (snr_holder < 9.70))
                {
                new_operational_speed = 36000000;
                }
        else if ((snr_holder >= 6.76) && (snr_holder < 8.86))
                {
                new_operational_speed = 24000000;
```

```
            }
        else if ((snr_holder >= 6.30) && (snr_holder < 6.76))
            {
            new_operational_speed = 18000000;
            }
        else if ((snr_holder >= 5.84) && (snr_holder < 6.30))
            {
            new_operational_speed = 12000000;
            }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
            {
            new_operational_speed = 9000000;
            }
        else
            {
            new_operational_speed = 6000000;
            }
        }
    else if (max_operational_speed == 48000000)
        {
        if (snr_holder >= 9.70)
            {
            new_operational_speed = max_operational_speed;
            }
        else if ((snr_holder >= 8.86) && (snr_holder < 9.70))
            {
            new_operational_speed = 36000000;
            }
        else if ((snr_holder >= 6.76) && (snr_holder < 8.86))
            {
            new_operational_speed = 24000000;
            }
        else if ((snr_holder >= 6.30) && (snr_holder < 6.76))
            {
            new_operational_speed = 18000000;
            }
        else if ((snr_holder >= 5.84) && (snr_holder < 6.30))
            {
            new_operational_speed = 12000000;
            }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
            {
            new_operational_speed = 9000000;
            }
        else
            {
            new_operational_speed = 6000000;
            }
        }
    else if (max_operational_speed == 36000000)
        {
        if (snr_holder >= 8.86)
            {
```

```
                new_operational_speed = max_operational_speed;
                }
        else if ((snr_holder >= 6.76) && (snr_holder < 8.86))
                {
                new_operational_speed = 24000000;
                }
        else if ((snr_holder >= 6.30) && (snr_holder < 6.76))
                {
                new_operational_speed = 18000000;
                }
        else if ((snr_holder >= 5.84) && (snr_holder < 6.30))
                {
                new_operational_speed = 12000000;
                }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
                {
                new_operational_speed = 9000000;
                }
        else
                {
                new_operational_speed = 6000000;
                }
        }
else if (max_operational_speed == 24000000)
        {
        if (snr_holder >= 6.76)
                {
                new_operational_speed = max_operational_speed;
                }
        else if ((snr_holder >= 6.30) && (snr_holder < 6.76))
                {
                new_operational_speed = 18000000;
                }
        else if ((snr_holder >= 5.84) && (snr_holder < 6.30))
                {
                new_operational_speed = 12000000;
                }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
                {
                new_operational_speed = 9000000;
                }
        else
                {
                new_operational_speed = 6000000;
                }
        }
else if (max_operational_speed == 18000000)
        {
        if (snr_holder >= 6.30)
                {
                new_operational_speed = max_operational_speed;
                }
        else if ((snr_holder >= 5.84) && (snr_holder < 6.30))
```

```
                {
                new_operational_speed = 12000000;
                }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
                {
                new_operational_speed = 9000000;
                }
        else
                {
                new_operational_speed = 6000000;
                }
        }
else if (max_operational_speed == 12000000)
        {
        if (snr_holder >= 5.84)
                {
                new_operational_speed = max_operational_speed;
                }
        else if ((snr_holder >= 5.38) && (snr_holder < 5.84))
                {
                new_operational_speed = 9000000;
                }
        else
                {
                new_operational_speed = 6000000;
                }
        }
else if (max_operational_speed == 9000000)
        {
        if (snr_holder >= 5.38)
                {
                new_operational_speed = max_operational_speed;
                }
        else
                {
                new_operational_speed = 6000000;
                }
        }
else
        {
        new_operational_speed = 6000000;
        }

/* Set the new data rate for the STA.                    */
operational_speed = new_operational_speed;
/* Report the operational speed of the WLAN using the new Statistic.   */
op_stat_write (operational_rate_handle, operational_speed);
op_stat_write (global_operational_rate_handle, operational_speed);
```

APPENDIX C. PACKET LOSS RATE-BASED RATE AGILITY OPNET CODE

This appendix contains the modifications to the *wlan_mac_11a* process model used to realize the packet loss rate-based dynamic data rate agility mechanism presented in Chapter IV. The preponderance of alterations are to the *wlan_prepare_frame_to_send* function. Those modifications are included in this appendix. Additional changes to *wlan_mac_11a* include:

- Initialization of the *max_operational_speed* variable and addition of the data rate statistic collection function in the *wlan_mac_svc_init* function.
- Incrementation of the *drop_counter* variable in the *wlan_frame_discard* function.

The block of code presented below is the first portion of the *wlan_prepare_frame_to_send* function that includes the rate agility mechanism additions. The remainder of the function remainsthe same. Comments indicating the nature of the code changes are included.

OPNET Code additions to *wlan_prepare_frame_to_send*
Bryan E. Braswell
March 2001

```
static void
wlan_prepare_frame_to_send (int frame_type)
	{
	char					msg_string [120];
	Packet*					hld_pkptr;
	Packet*					seg_pkptr;
	int					dest_addr, src_addr;
	int					protocol_type = -1;
	int					tx_datapacket_size;
	int					type;
	char					error_string [512];
	int					outstrm_to_phy;
	double					duration, mac_delay;
	WlanT_Data_Header_Fields*		pk_dhstruct_ptr;
	WlanT_Control_Header_Fields*		pk_chstruct_ptr;
	Packet*					wlan_transmit_frame_ptr;

	/* 802.11a Model Addition */
	/* Add a variable to keep track of the data rate so it can be passed to the pipeline stages. */
```

```
int                                         rate_holder;

/*  802.11a Model Addition  */
/* The control frame transmission rate depends on the given data rate.  */
/* Adapted from the Philips Lab 802.11a model (dated 11/15/00).            */
double              control_frame_speed;    /* Speed for control frames. */
int                 next_frag_length;       /* Length of the next fragment (in bits). */
int                 MPDU_size;              /* MPDU length (in bits). */

/* Dropped Packet Data Rate Agility Mechansim Addition.  */
double        window;
double        drop_rate;
double        new_operational_speed;
double        steady_state_timer;

/** Prepare frames to transmit by setting appropriate fields in the         **/
/** packet format for Data,Cts,Rts or Ack.  If data or Rts packet needs **/
/** to be retransmitted then the older copy of the packet is resent.        **/
FIN (wlan_prepare_frame_to_send (int frame_type));

outstrm_to_phy = LOW_LAYER_OUT_STREAM_CH1;

/*  802.11a Model Addition  */
rate_holder = 1;

/*  802.11a Model Addition  */
/* Compute the control frame speed based on the operational data rate.  */
/* Adapted from the Philips Lab 802.11a model code (dated 11/15/00).  */
control_frame_speed = control_speed (operational_speed);

/* Dropped Packet Data Rate Agility Mechansim Addition.     */
/* Compute the time window size.                            */
window = current_time - time_counter;
steady_state_timer = current_time - steady_state_counter;

/* Compute the number of packets dropped per unit time in this window.  */
drop_rate = drop_counter / window;

/* Based on the dropped packet rate, adjust the data rate if necessary.  */
if (drop_rate > 0.11249)
        {
        if (operational_speed == 54000000)
                {
                new_operational_speed = 48000000;
                steady_state_counter = current_time;
                }
        else if (operational_speed == 48000000)
                {
                new_operational_speed = 36000000;
                steady_state_counter = current_time;
                }
        else if (operational_speed == 36000000)
                {
```

```
                new_operational_speed = 24000000;
                steady_state_counter = current_time;
                }
        else if (operational_speed == 24000000)
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if (operational_speed == 18000000)
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if (operational_speed == 12000000)
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else
                {
                new_operational_speed = 6000000;
                steady_state_counter = current_time;
                }
        }
    else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 54000000))
        {
        new_operational_speed = 54000000;
        steady_state_counter = current_time;
        }
    else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 48000000))
        {
        new_operational_speed = 54000000;
        steady_state_counter = current_time;
        }
    else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 36000000))
        {
        new_operational_speed = 48000000;
        steady_state_counter = current_time;
        }
    else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 24000000))
        {
        new_operational_speed = 36000000;
        steady_state_counter = current_time;
        }
    else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 18000000))
        {
        new_operational_speed = 24000000;
        steady_state_counter = current_time;
```

```
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 12000000))
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 9000000))
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
54000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 48000000))
                {
                new_operational_speed = 48000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 36000000))
                {
                new_operational_speed = 48000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 24000000))
                {
                new_operational_speed = 36000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 18000000))
                {
                new_operational_speed = 24000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 12000000))
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 9000000))
                {
                new_operational_speed = 12000000;
```

```
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
48000000) && (operational_speed == 6000000))
            {
            new_operational_speed = 9000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 36000000))
            {
            new_operational_speed = 36000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 24000000))
            {
            new_operational_speed = 36000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 18000000))
            {
            new_operational_speed = 24000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 12000000))
            {
            new_operational_speed = 18000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 9000000))
            {
            new_operational_speed = 12000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
36000000) && (operational_speed == 6000000))
            {
            new_operational_speed = 9000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
24000000) && (operational_speed == 24000000))
            {
            new_operational_speed = 24000000;
            steady_state_counter = current_time;
            }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
24000000) && (operational_speed == 18000000))
            {
```

```
                new_operational_speed = 24000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 24000000) && (operational_speed == 12000000))
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 24000000) && (operational_speed == 9000000))
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 24000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 18000000) && (operational_speed == 18000000))
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 18000000) && (operational_speed == 12000000))
                {
                new_operational_speed = 18000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 18000000) && (operational_speed == 9000000))
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 18000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 12000000) && (operational_speed == 12000000))
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed == 12000000) && (operational_speed == 9000000))
```

```
                {
                new_operational_speed = 12000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
12000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
9000000) && (operational_speed == 9000000))
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
9000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 9000000;
                steady_state_counter = current_time;
                }
        else if ((steady_state_timer > 10.0) && (drop_rate == 0.0) && (max_operational_speed ==
6000000) && (operational_speed == 6000000))
                {
                new_operational_speed = 6000000;
                steady_state_counter = current_time;
                }
        else
                {
                new_operational_speed = operational_speed;
                }

        /* Now assign the new data rate to the station.    */
        operational_speed = new_operational_speed;

        /* Now we need to check the window size and adjust if the window has become too big.    */
        /* The window size utilized here is 2 seconds.                                           */
        if (window > 1.0)
                {
                drop_counter = 0;
                time_counter = current_time;
                }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX D. DRA_SNR_11A PIPELINE STAGE

This appendix presents the source code for the *dra_snr_11a* pipeline stage used in conjunction with the SNR-based rate agility mechanism OPNET model. This code is a modified version of the default *dra_snr* pipeline stage. Comments indicating the nature of the code changes are included.

dra_snr_11a Pipeline Stage Bryan E. Braswell March 2001

```
/* dra_snr.ps.c */
/* Default Signal-to-Noise-Ratio (SNR) model for radio link Transceiver Pipeline */

/***********************************************/
/*                  Copyright (c) 1993-2000          */
/*                by OPNET Technologies, Inc.        */
/*                (A Delaware Corporation)           */
/*          3400 International Drive, N.W.           */
/*                Washington, D.C., U.S.A.           */
/*                      All Rights Reserved.         */
/***********************************************/

#include "opnet.h"
#include <math.h>

#if defined (__cplusplus)
extern "C"
#endif
void
dra_snr (Packet * pkptr)
	{
	double		bkg_noise, accum_noise, rcvd_power;
	double		s_n_r;

	/** Compute the signal-to-noise ratio for the given packet. **/
	FIN (dra_snr (pkptr));

	/* Get the packet's received power level. */
	rcvd_power = op_td_get_dbl (pkptr, OPC_TDA_RA_RCVD_POWER);

	/* Get the packet's accumulated noise levels calculated by the */
	/* interference and background noise stages. */
	accum_noise = op_td_get_dbl (pkptr, OPC_TDA_RA_NOISE_ACCUM);
	bkg_noise = op_td_get_dbl (pkptr, OPC_TDA_RA_BKGNOISE);

	/* Compute the SNR.  */
	s_n_r = (10.0 * log10 (rcvd_power / (accum_noise + bkg_noise)));
```

```
/* Assign the SNR in dB. */
op_td_set_dbl (pkptr, OPC_TDA_RA_SNR, s_n_r);

/* Place the SNR in the Link SNR packet header field for use   */
/* in the SNR-based rate agility mechanism.                    */
op_pk_nfd_set (pkptr, "Link SNR", s_n_r);

/* Set field indicating the time at which SNR was calculated. */
op_td_set_dbl (pkptr, OPC_TDA_RA_SNR_CALC_TIME, op_sim_time ());

FOUT;
}
```

APPENDIX E. DRA_TXDEL_11A PIPELINE STAGE

This appendix presents the source code for the *dra_txdel_11a* pipeline stage used in conjunction with the baseline 802.11a OPNET model. This code is a modified version of the default *dra_txdel* pipeline stage. Comments indicating the nature of the code changes are included.

dra_txdel_11a Pipeline Stage
Bryan E. Braswell
March 2001

```
/* dra_txdel.ps.c */
/* Default transmission delay model for radio link Transceiver Pipeline */

/**********************************************/
/*              Copyright (c) 1993-2000          */
/*           by OPNET Technologies, Inc.         */
/*            (A Delaware Corporation)           */
/*     3400 International Drive, N.W.            */
/*            Washington, D.C., U.S.A.           */
/*                  All Rights Reserved.         */
/**********************************************/

#include "opnet.h"

#if defined (__cplusplus)
extern "C"
#endif
void
dra_txdel (Packet * pkptr)
	{
	int		pklen;
	double		tx_drate, tx_delay;
	int		rate_index;

	/** Compute the transmission delay associated with the	**/
	/** transmission of a packet over a radio link.		**/
	FIN (dra_txdel (pkptr));

	/* Obtain the transmission rate of that channel. */
	//tx_drate = op_td_get_dbl (pkptr, OPC_TDA_RA_TX_DRATE);

	/* Change for the 802.11a model.  */
	/* The transmission data rate is variable based on the  */
	/* use of control or data frames. So, the Rate packet	*/
	/* field is used to determine the data rate for the calculation.  */
	op_pk_nfd_access (pkptr, "Rate", &rate_index);
```

```
if (rate_index == 1)
        {
        tx_drate = 6000000;
        }
else if (rate_index == 2)
        {
        tx_drate = 9000000;
        }
else if (rate_index == 3)
        {
        tx_drate = 12000000;
        }
else if (rate_index == 4)
        {
        tx_drate = 18000000;
        }
else if (rate_index == 5)
        {
        tx_drate = 24000000;
        }
else if (rate_index == 6)
        {
        tx_drate = 36000000;
        }
else if (rate_index == 7)
        {
        tx_drate = 48000000;
        }
else
        {
        tx_drate = 54000000;
        }

/* Obtain length of packet. */
pklen = op_pk_total_size_get (pkptr);

/* Compute time required to complete transmission of packet. */
tx_delay = pklen / tx_drate;

/* Place transmission delay result in packet's */
/* reserved transmission data attribute. */
op_td_set_dbl (pkptr, OPC_TDA_RA_TX_DELAY, tx_delay);

FOUT;
}
```

LIST OF REFERENCES

1. Choi, S., Philips Research Labs, OPNET Model of the 802.11a Protocol, 15 November, 2000.

2. Institute of Electrical and Electronics Engineers, 802.11, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 20 August 1999.

3. Institute of Electrical and Electronics Engineers, 802.11b, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher-Speed Physical Layer Extension in the 2.4 GHz Band*, 16 September 1999.

4. Institute of Electrical and Electronics Engineers, 802.11a, *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: High-Speed Physical Layer in the 5 GHz Band*, 16 September 1999.

5. European Telecommunications Standards Institute, EN 300 652 V1.2.1, *High Performance Radio Local Area Network (HIPERLAN) Type 1; Functional Specification*, July 1998.

6. Awater, G., van Nee, R., Morikura, M., Takanashi, H., Webster, M., and Halford, K. W., "New High-Rate Wireless LAN Standards," *IEEE Communications Magazine*, Vol 37 Issue 12, pp. 82-88, December 1999.

7. Stallings, W., *Data and Computer Communications*, Prentice Hall, 1997.

8. 3Com Corporation, "What's New in Wireless LANs: The IEEE 802.11b Standard." [http://www.3com.com/technology/tech_net/white_papers/503072a.html]. October 2000.

9. Moore, C., "Faster Wireless LANs on Tap," *Infoworld.com.* [http://www.infoworld.com/articles/hn/xml/00/09/15/000915hnradiata.xml]. 15 September 2000.

10. European Telecommunications Standards Institute, ETSI TR 101 683 V1.1.1, *Broadband Radio Access Networks (BRAN); HIPERLAN Type 2; System Overview*, February 2000.

11. Johnsson, M., "HiperLAN/2 – The Broadband Radio Transmission Technology Operating in the 5 GHz Frequency Band," *HiperLAN/2 Global Forum*, 1999.

12. Fisher, C., "The Wireless Market: Growth Hinges on the Right Solution," Radiata, Inc., 15 September 2000.

13. Radiata, Inc. Press Release, "Radiata First to Deliver High-performance, Low-Cost Wireless Networking Solution," 15 September 2000.

14. Atheros Communications AR5000 Wireless LAN Solution Product Specification, "Next-Generation Wireless LAN in Mainstream CMOS Technology," 2000.

15. Bing, B., *High-Speed Wireless ATM and LANs*, Artech House, 2000.

16. O'Hara, B., Petrick, A., *The IEEE 802.11 Handbook: A Designer's Companion*, IEEE Press, December 1999.

17. Prasad, R., van Nee, R., *OFDM For Wireless Multimedia Communications*, Artech House, 2000.

18. OPNET Modeler Version 7.0B Online Documentation, OPNET Technologies, Inc., 16 May 2000.

19. OPNET Tutorial, "Introduction to OPNET Modeler," OPNETWORK 2000, 28 August 2000.

20. Tan, K. C., *Development and Simulation of the 802.11a Physical Layer to Study the the Effects of Multipath on its Performance*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 2001.

21. Electronic Mail Message from Haug, N. B., Lucent Technical Support Center, Subject: WaveLAN Questions, 26 February 2001.

22. Perez-Vega, C. and Garcia, J. L. G., "A Simple Approach to a Statistical Path Loss model for Indoor Communications," *Proceedings of the 27th European Microwave Conference and Exhibition*, pp. 617-623, 08-12 September 1997.

23. Durgin, G. D., Rappaport, T. S., and Xu, H., "Partition-Based Path Loss Analysis for In-Home and Residential Areas at 5.85 GHz," *Proceedings of the 1998 Global Telecommunications Conference*, Vol. 2, pp. 904-909, 08-12 November 1998.

24. Berg, J. E. and Medbo, J., "Simple and Accurate Path Loss Modeling at 5 GHz in Indoor Environments with Corridors," *Proceedings of the 2000 Vehicular technology Conference*, Vol. 1, pp. 30-36, 24-28 September 2000.

25. Lucent Technologies, Inc., "ORiNOCO PC Card Specifications," 2000.

26. Lucent Technologies, Inc., "ORiNOCO Manager Suite User's Guide," August 2000.

27. Chow, C. C. and Leung, V. C. M., "Performance of IEEE 802.11 Medium Access Control Protocol Over a Wireless Local Area Network with Distributed Radio Bridges,"

28. Aironet Wireless Communications, Inc., "PCI & ISA 4800/4500 Wireless LAN Adapter Specifications," 1998.

29. Electronic Mail Message from Chesson, G., Atheros Communications, Inc., Subject: Rate Adaptation, 30 January 2001.

30. Rappaport, T. S., *Wireless Communications*, Prentice Hall, January 1996.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Fort Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, California 93943-5101

3. Commanding Officer, Naval Information Warfare Activity 1
   9800 Savage Road
   Fort George G. Meade, MD 20755-6000

4. Director, CNO Strategic Studies Group 1
   686 Cushing Road
   Newport, RI 02841-1207

5. Chairman, Code EC 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

6. Professor John McEachen, Code EC/Mj 2
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

7. Professor Murali Tummala, Code EC/Tu 1
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California 93943-5121

8. Dr. Sunghyun Choi 1
   Video Communications Department
   Philips Research Labs
   345 Scarborough Road
   Briarcliff Manor, NY 10510

9. Mr. Matthew Sherman 1
   PTSM – Communications Technology Research
   AT&T Labs – Shannon Laboratory
   Room 3K18, Building 104
   180 Park Avenue
   Florham Park, NJ 07932-0971

10. LT Bryan E. Braswell, USN 1
7902 Red Globe Court
Severn, MD 21144

11. LTC (Ret) Billie E. Braswell, USA 1
12215 Dell Way
Fredericksburg, VA 22407

12. Department of Defense 1
ATTN: R531
9800 Savage Rd.
Fort George G. Meade, MD 20755

13. Office of Naval Research... 1
800 N. Quincy St.
Arlington, VA 22217-5660

14. SPAWAR Systems Center 1
ATTN: D855
53560 Hull St.
San Diego, CA 92152-5001

15. Naval Information Warfare Analysis Center 1
Code 5707
4555 Overlook Ave. SW
Washington, DC 20375-5707

16. Atheros Communications, Inc.. 1
ATTN: Dr. Greg Chesson
529 Almanor Ave.
Sunnyvale, CA 94085-3512